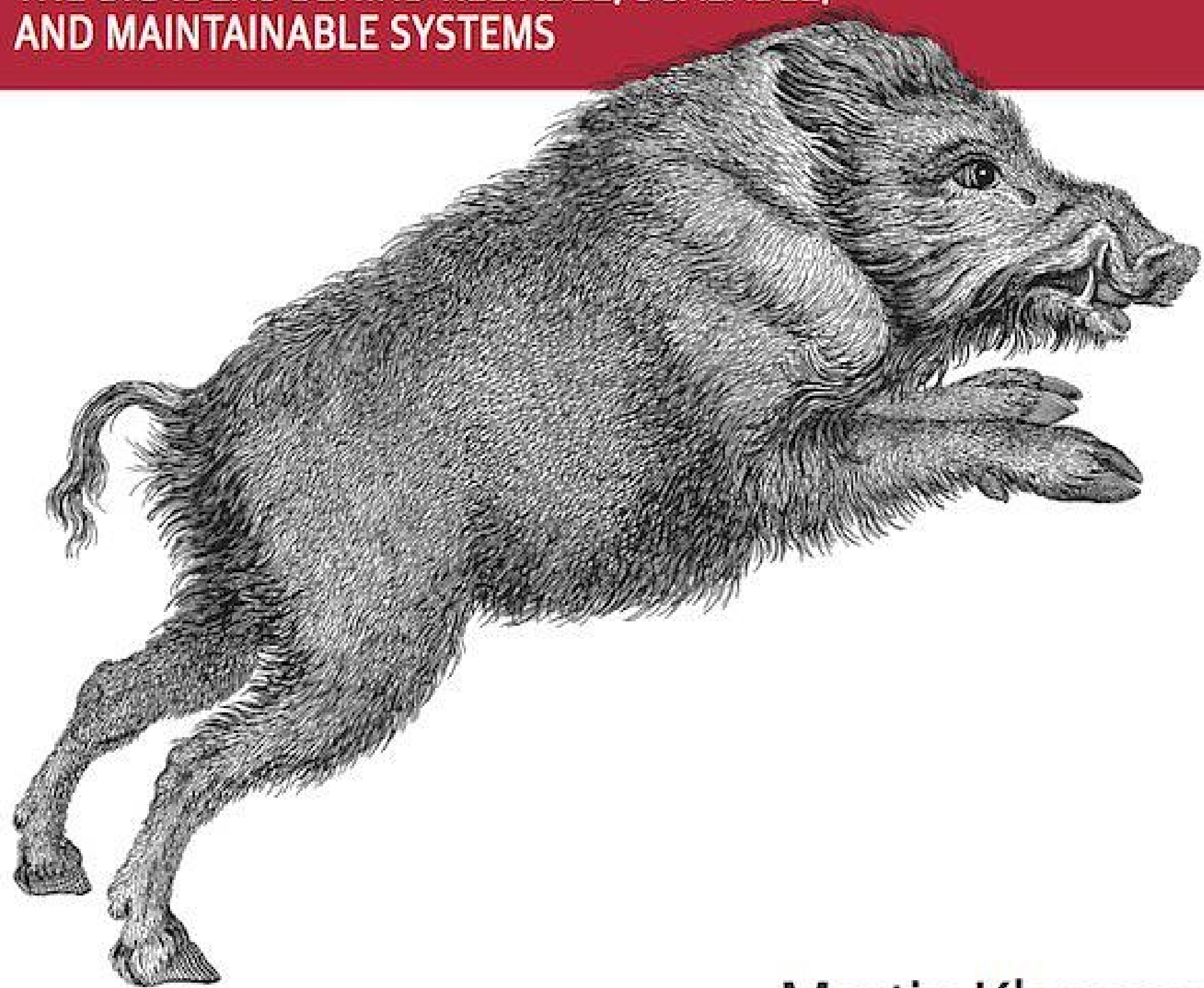
O'REILLY®

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS



Martin Kleppmann

目錄

设计数据密集型应用 - 中文翻译

> 来源：Vonng/ddia

- 作者： Martin Kleppmann
- 原书名称：《Designing Data-Intensive Application》
- 译者：冯若航 （fengruohang@outlook.com ）
- Gitbook地址：ddia-cn（需要科学上网）
- 建议使用Typora或Gitbook以获取最佳阅读体验。

法律声明

从原作者处得知，已经有简体中文的翻译计划，将于2018年末完成。

译者纯粹出于学习目的与个人兴趣翻译本书，不追求任何经济利益。

译者保留对此版本译文的署名权，其他权利以原作者和出版社的主张为准。

本译文只供学习研究参考之用，不得公开传播发行或用于商业用途。有能力阅读英文书籍者请购买正版支持。

译序

> 不懂数据库的全栈工程师不是好架构师
>
> —— Vonng

现今，尤其是在互联网领域，大多数应用都属于数据密集型应用。本书从底层数据结构到顶层架构设计，将数据系统设计中的精髓娓娓道来。其中的宝贵经验无论是对架构师，DBA、还是后端工程师、甚至产品经理都会有帮助。

这是一本理论结合实践的书，书中很多问题，译者在实际场景中都曾遇到过，读来让人击节扼腕。如果能早点读到这本书，该少走多少弯路啊！

这也是一本深入浅出的书，讲述概念的来龙去脉而不是卖弄定义，介绍事物发展演化历程而不是事实堆砌，将复杂的概念讲述的浅显易懂，但又直击本质不失深度。每章最后的引用质量非常好，是深入学习各个主题的绝佳索引。

本书为数据系统的设计、实现、与评价提供了很好的概念框架。读完并理解本书内容后，读者可以轻松看破大多数的技术忽悠，与技术砖家撕起来虎虎生风 。

这是2017年译者读过最好的一本技术类书籍，这么好的书没有中文翻译，实在是遗憾。某不才，愿为先进技术文化的传播贡献一分力量。既可以深入学习有趣的技术主题，又可以锻炼中英文语言文字功底，何乐而不为？

不过翻译，尤其是精翻，确实是一件极其耗费心血的工作。没有什么经济收益，只有纯粹的兴趣，欢迎有兴趣的朋友一起加入。

前言

> 在我们的社会中，技术是一种强大的力量。数据、软件、通信可以用于坏的方面：不公平的阶级固化，损害公民权利，保护既得利益集团。但也可以用于好的方面：让底层人民发出自己的声音，让每个人都拥有机会，避免灾难。本书献给所有将技术用于善途的人们。

> 计算是一种流行文化，流行文化鄙视历史。 流行文化关乎个体身份和参与感，但与合作无关。流行文化活在当下，也与过去和未来无关。 我认为大部分（为了钱）编写代码的人就是这样的， 他们不知道自己的文化来自哪里。
>
> ——阿兰·凯接受Dobb博士的杂志采访时（2012年）

目录

第三部分：衍生数据

术语表

后记

翻译计划

* 机翻：只在乎结构：梳理文章结构、图片、引用、备注。
* 初翻：保证经完全理解本章内容，人工修复显著的错误，重新组织语言。
* 精校：阅读相关领域文献书籍，确定术语的最终译法，修复格式瑕疵，着力信达雅。

精翻可以看，机翻基本没法看，初翻对于业内人士能凑合看。

章节	进度	锁定
序言	初翻	
第一部分：数据系统基础	精翻	
第一章：可靠性、可扩展性、可维护性	精翻	
第二章：数据模型与查询语言	初翻	@ jiajiadebug
第三章：存储与检索	初翻 40%	Vonng
第四章：编码与演化	初翻	
第二部分：分布式数据	精翻	
第五章：复制	精翻 30%	Vonng
第六章：分区	初翻	
第七章：事务	精翻 60%	Vonng
第八章：分布式系统中的问题	初翻	
第九章：一致性与共识	初翻	
第三部分：衍生数据	精翻	
第十章：批处理	草翻	
第十一章：流处理	草翻	
第十二章：数据系统的未来	草翻	
术语表	-	
后记	机翻	

CONTRIBUTION

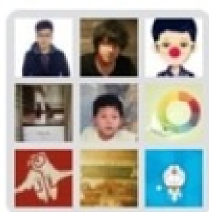
欢迎贡献

整章的翻译、精校请使用PR，将列入译者署名，少量词法、语法、译法问题请使用ISSUE，将列入致谢。

All contribution will give proper credit. 贡献者需要同意法律声明所叙内容。

1. 序言初翻修正 by @seagullbird
2. 第一章语法标点校正 by @nevertiree
3. 第六章部分校正 by @MuAlex
4. 第一部分前言，ch2校正 by @jiajiadebug

译读者交流微信群





LICENSE

CC-BY 4.0

序言

如果近几年从业于软件工程，特别是服务器端和后端系统开发，那么您很有可能已经被大量关于数据存储和处理的时髦词汇轰炸过了： NoSQL！大数据！Web-Scale！分片！最终一致性！ACID！ CAP定理！云服务！MapReduce！实时！

在最近十年中，我们看到了很多有趣的进展，关于数据库，分布式系统，以及在此基础上构建应用程序的方式。这些进展有着各种各样的驱动力：

- 谷歌，雅虎，亚马逊，脸书，领英，微软和推特等互联网公司正在和巨大的流量/数据打交道，这迫使他们去创造能有效应对如此规模的新工具。
- 企业需要变得敏捷，需要低成本地检验假设，需要通过缩短开发周期和保持数据模型的灵活性，快速地响应新的市场洞察。
- 免费和开源软件变得非常成功，在许多环境中比商业软件和定制软件更受欢迎。
- 处理器主频几乎没有增长，但是多核处理器已经成为标配，网络也越来越快。这意味着并行化程度只增不减。
- 即使您在一个小团队中工作，现在也可以构建分布在多台计算机甚至多个地理区域的系统，这要归功于譬如亚马逊网络服务（AWS）等基础设施即服务（IaaS）概念的践行者。
- 许多服务都要求高可用，因停电或维护导致的服务不可用，变得越来越难以接受。

数据密集型应用（**data-intensive applications**）正在通过使用这些技术进步来推动可能性的边界。一个应用被称为数据密集型的，如果数据是其主要挑战（数据量，数据复杂度或数据变化速度）—— 与之相对的是计算密集型，即处理器速度是其瓶颈。

帮助数据密集型应用存储和处理数据的工具与技术，正迅速地适应这些变化。新型数据库系统（“NoSQL”）已经备受关注，而消息队列，缓存，搜索索引，批处理和流处理框架以及相关技术也非常重要。很多应用组合使用这些工具与技术。

这些生意盎然的时髦词汇体现出人们对新的可能性的热情，这是一件好事。但是作为软件工程师和架构师，如果要开发优秀的应用，我们还需要对各种层出不穷的技术及其利弊权衡有精准的技术理解。为了获得这种洞察，我们需要深挖时髦词汇背后的内容。

幸运的是，在技术迅速变化的背后总是存在一些持续成立的原则，无论您使用了特定工具的哪个版本。如果您理解了这些原则，就可以领会这些工具的适用场景，如何充分利用它们，以及如何避免其中的陷阱。这正是本书的初衷。

本书的目标是帮助您在飞速变化的数据处理和数据存储技术大观园中找到方向。本书并不是某个特定工具的教程，也不是一本充满枯燥理论的教科书。相反，我们将看到一些成功数据系统的样例：许多流行应用每天都要在生产中会满足可扩展性、性能、以及可靠性的要求，而这些技术构成了这些应用的基础。

我们将深入这些系统的内部，理清它们的关键算法，讨论背后的原则和它们必须做出的权衡。在这个过程中，我们将尝试寻找思考数据系统的有效方式 —— 不仅关于它们如何工作，还包括它们为什么以这种方式工作，以及哪些问题是我们需要问的。

阅读本书后，你能很好地决定哪种技术适合哪种用途，并了解如何将工具组合起来，为一个良好应用架构奠定基础。本书并不足以使你从头开始构建自己的数据库存储引擎，不过幸运的是这基本上很少有必要。你将获得对系统底层发生事情的敏锐直觉，这样你就有能力推理它们的行为，做出优秀的设计决策，并追踪任何可能出现的问题。

本书的目标读者

如果你开发的应用具有用于存储或处理数据的某种服务器/后端系统，而且使用网络（例如，Web应用，移动应用或连接到互联网的传感器），那么本书就是为你准备的。

本书是为软件工程师，软件架构师，以及喜欢写代码的技术经理准备的。如果您需要对所从事系统的架构做出决策 —— 例如您需要选择解决某个特定问题的工具，并找出如何最好地使用这些工具，那么这本书对您尤有价值。但即使你无法选择你的工具，本书仍将帮助你更好地了解所使用工具的长处和短处。

您应当具有一些开发Web应用或网络服务的经验，且应当熟悉关系型数据库和SQL。任何您了解的非关系型数据库和其他与数据相关工具都会有所帮助，但不是必需的。对常见网络协议如TCP和HTTP的大概理解是有帮助的。编程语言或框架的选择对阅读本书没有任何不同影响。

如果以下任意一条对您为眞，你会发现这本书很有价值：

- 您想了解如何使数据系统可扩展，例如，支持拥有数百万用户的Web或移动应用。
- 您需要提高应用程序的可用性（最大限度地减少停机时间），保持稳定运行。
- 您正在寻找使系统在长期运行过程易于维护的方法，即使系统规模增长，需求与技术也发生变化。
- 您对事物的运作方式有着天然的好奇心，并且希望知道一些主流网站和在线服务背后发生的事情。这本书打破了各种数据库和数据处理系统的内幕，探索这些系统设计中的智慧是非常有趣的。

有时在讨论可扩展的数据系统时，人们会说：“你又不在谷歌或亚马逊，别操心可扩展性了，直接上关系型数据库”。这个陈述有一定的道理：为了不必要的扩展性而设计程序，不仅会浪费不必要的精力，并且可能会把你锁死在一个不灵活的设计中。实际上这是一种“过早优化”的形式。不过，选择合适的工具确实很重要，而不同的技术各有优缺点。我们将看到，关系数据库虽然很重要，但绝不是数据处理的终章。

本书涉及的领域

本书并不会尝试告诉读者如何安装或使用特定的软件包或API，因为已经有大量文档给出了详细的使用说明。相反，我们会讨论数据系统的基石——各种原则与利弊权衡，并探讨了不同产品所做出的不同设计决策。

在电子书中包含了在线资源全文的链接。所有链接在出版时都进行了验证，但不幸的是，由于网络的自然规律，链接往往会频繁地破损。如果您遇到链接断开的情况，或者正在阅读本书的打印副本，可以使用搜索引擎查找参考文献。对于学术论文，您可以在Google学术中搜索标题，查找可以公开获取的PDF文件。或者，您也可以在 https://github.com/ept/ddia-references 中找到所有的参考资料，我们在那儿维护最新的链接。

我们主要关注的是数据系统的架构（**architecture**），以及它们被集成到数据密集型应用中的方式。本书没有足够的空间覆盖部署，运维，安全，管理等领域 —— 这些都是复杂而重要的主题，仅仅在本书中用粗略的注解讨论这些对它们很不公平。每个领域都值得用单独的书去讲。

本书中描述的许多技术都被涵盖在大数据（**Big Data**）这个时髦词的范畴中。然而“大数据”这个术语被滥用，缺乏明确定义，以至于在严肃的工程讨论中没有用处。这本书使用歧义更小的术语，如“单节点”之于”分布式系统“，或”在线/交互式系统“之于”离线/批处理系统“。

本书对自由和开源软件（FOSS）有一定偏好，因为阅读，修改和执行源码是了解一样东西详细工作原理的好方法。开放的平台也可以降低供应商垄断的风险。然而在适当的情况下，我们也会讨论专利软件（闭源软件，软件即服务 SaaS，或一些在文献中描述过但未公开发行的公司内部软件）。

本书纲要

本书分为三部分：

1. 在第一部分中，我们会讨论设计数据密集型应用所赖的基本思想。我们从第1章开始，讨论我们实际要达到的目标：可靠性，可扩展性和可维护性；我们该如何思考这些概念；以及如何实现它们。在第2章中，我们比较了几种不同的数据模型和查询语言，看看它们如何适用于不同的场景。在第3章中将讨论存储引擎：数据库如何在磁盘上摆放数据，以便能高效地再次找到它。第4章转向数据编码（序列化），以及随时间演化的模式。

2. 在第二部分中，我们从讨论存储在一台机器上的数据转向讨论分布在多台机器上的数据。这对于可扩展性通常是必需的，但带来了各种独特的挑战。我们首先讨论复制（第5章），分区/分片（第6章）和事务（第7章）。然后我们将探索关于分布式系统问题的更多细节（第8章），以及在分布式系统中实现一致性与共识意味着什么（第9章）。

3. 在第三部分中，我们讨论那些从其他数据集衍生出一些数据集的系统。衍生数据经常出现在异构系统中：当没有单个数据库可以把所有事情都做的很好时，应用需要集成几种不同的数据库，缓存，索引等。在第10章中我们将从一种衍生数据的批处理方法开始，

然后在此基础上建立在第11章中讨论的流处理。最后，在第12章中，我们将所有内容汇总，讨论在将来构建可靠，可伸缩和可维护的应用程序的方法。

参考文献与延伸阅读

本书中讨论的大部分内容已经在其它地方以某种形式出现过了 —— 会议演示文稿，研究论文，博客文章，代码，BUG跟踪器，邮件列表，以及工程习惯中。本书总结了不同来源资料中最重要的想法，并在文本中包含了指向原始文献的链接。 如果你想更深入地探索一个领域，那么每章末尾的参考文献都是很好的资源，其中大部分可以免费在线获取。

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interac- tive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Pro- fessional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit http://oreilly.com/safari.

致谢

本书融合了学术研究和工业实践的经验，融合并系统化了大量其他人的想法与知识。在计算领域，我们往往会被各种新鲜花样所吸引，但我认为前人完成的工作中，有太多值得我们学习的地方了。本书有800多处引用：文章，博客，讲座，文档等，对我来说这些都是宝贵的学习资源。我非常感谢这些材料的作者分享他们的知识。

我也从与人交流中学到了很多东西，很多人花费了宝贵的时间与我讨论想法并耐心解释。特别感谢 Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Hel- land, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha

Narkhede, Neha Narula, Cathy O’Neil, Onora O’Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden, 以及 Brett Wooldridge.

更多人通过审阅草稿并提供反馈意见在本书的创作过程中做出了无价的贡献。我要特别感谢Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes, 以及Ben Summers。当然对于本书中的任何遗留错误或难以接受的见解，我都承担全部责任。

为了帮助这本书落地，并且耐心地处理我缓慢的写作和不寻常的要求，我要对编辑Marie Beaugureau，Mike Loukides，Ann Spencer和O'Reilly的所有团队表示感谢。我要感谢Rachel Head帮我找到了合适的术语。我要感谢Alastair Beresford，Susan Goodhue，Neha Narkhede和Kevin Scott，在其他工作事务之外给了我充分地创作时间和自由。

特别感谢Shabbir Diwan和Edie Freedman，他们非常用心地为各章配了地图。他们提出了不落俗套的灵感，创作了这些地图，美丽而引人入胜，真是太棒了。

最后我要表达对家人和朋友们的爱，没有他们，我将无法走完这个将近四年的写作历程。你们是最棒的。

第一部分 数据系统基础

本书前四章介绍了数据系统底层的基础概念，无论是在单台机器上运行的单点数据系统，还是分布在多台机器上的分布式数据系统都适用。

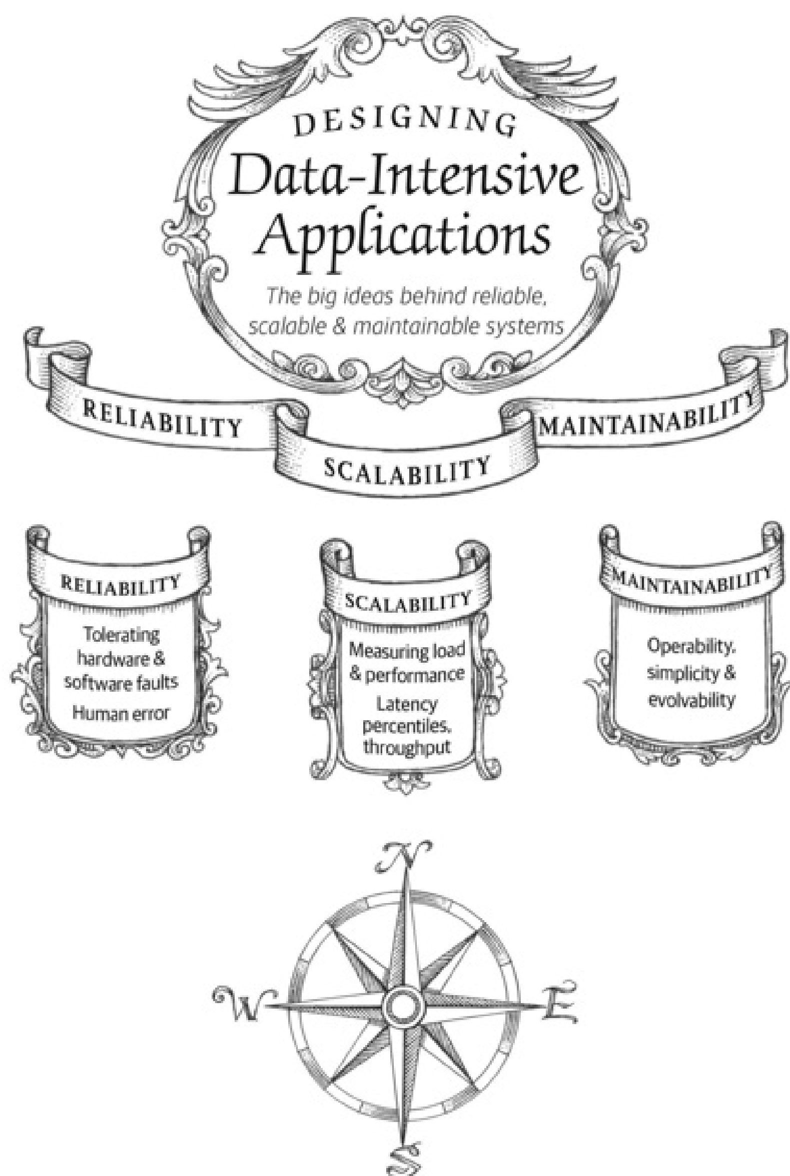
1. 第一章将介绍本书使用的术语和方法。可靠性，可扩展性和可维护性 ，这些词汇到底意味着什么？如何实现这些目标？
2. 第二章将对几种不同的数据模型和查询语言进行比较。从程序员的角度看，这是数据库之间最明显的区别。不同的数据模型适用于不同的应用场景。
3. 第三章将深入存储引擎内部，研究数据库如何在磁盘上摆放数据。不同的存储引擎针对不同的负载进行优化，选择合适的存储引擎对系统性能有巨大影响。
4. 第四章将对几种不同的 数据编码进行比较。特别研究了这些格式在应用需求经常变化、模式需要随时间演变的环境中表现如何。

第二部分将专门讨论在分布式数据系统中特有的问题。

目录

上一章	目录	下一章
序言	设计数据密集型应用	第一章：可靠性、可扩展性、可维护性

第一章：可靠性，可扩展性，可维护性



> 互联网做得太棒了，以至于大多数人将它看作像太平洋这样的自然资源，而不是什么人工产物。上一次出现这种大规模且无差错的技术， 你还记得是什么时候吗？
>
> ——阿兰·凯在接受Dobb博士杂志采访时说（2012年）

[TOC]

现今很多应用程序都是数据密集型（**data-intensive**）的，而非计算密集型（**compute-intensive**）的。因此CPU很少成为这类应用的瓶颈，更大的问题通常来自数据量、数据复杂性、以及数据的变更速度。

数据密集型应用通常由标准组件构建而成，标准组件提供了很多通用的功能；例如，许多应用程序都需要：

数据库（***database***）

 存储数据，以便自己或其他应用程序之后能再次找到

缓存（***cache***）

 记住开销昂贵操作的结果，加快读取速度

搜索索引（***search indexes***）

 允许用户按关键字搜索数据，或以各种方式对数据进行过滤

流处理（***stream processing***）

 向其他进程发送消息，进行异步处理

批处理（***batch processing***）

 定期处理累积的大批量数据

如果这些功能听上去平淡无奇，那是因为这些数据系统（**data system**）是非常成功的抽象：我们一直不假思索地使用它们并习以为常。绝大多数工程师不会幻想从零开始编写存储引擎，因为在开发应用时，数据库已经是足够完美的工具了。

但现实没有这么简单。不同的应用有着不同的需求，因而数据库系统也是百花齐放，有着各式各样的特性。实现缓存有很多种手段，创建搜索索引也有好几种方法，诸如此类。因此在开发应用前，我们依然有必要先弄清楚最适合手头工作的工具和方法。而且当单个工具解决不了你的问题时，组合使用这些工具可能还是有些难度的。

本书将是一趟关于数据系统原理、实践与应用的旅程，并讲述了设计数据密集型应用的方法。我们将探索不同工具之间的共性与特性，以及各自的实现原理。

本章将从我们所要实现的基础目标开始：可靠、可扩展、可维护的数据系统。我们将澄清这些词语的含义，概述考量这些目标的方法。并回顾一些后续章节所需的基础知识。在接下来的章节中我们将抽丝剥茧，研究设计数据密集型应用时可能遇到的设计决策。

关于数据系统的思考

我们通常认为，数据库、消息队列、缓存等工具分属于几个差异显著的类别。虽然数据库和消息队列表面上有一些相似性——它们都会存储一段时间的数据——但它们有迥然不同的访问模式，这意味着迥异的性能特征和实现手段。

那我们为什么要把这些东西放在数据系统（**data system**）的总称之下混为一谈呢？

近些年来，出现了许多新的数据存储工具与数据处理工具。它们针对不同应用场景进行优化，因此不再适合生硬地归入传统类别【1】。类别之间的界限变得越来越模糊，例如：数据存储可以被当成消息队列用（Redis），消息队列则带有类似数据库的持久保证（Apache Kafka）。

其次，越来越多的应用程序有着各种严格而广泛的要求，单个工具不足以满足所有的数据处理和存储需求。取而代之的是，总体工作被拆分成一系列能被单个工具高效完成的任务，并通过应用代码将它们缝合起来。

例如，如果将缓存（应用管理的缓存层，Memcached或同类产品）和全文搜索（全文搜索服务器，例如Elasticsearch或Solr）功能从主数据库剥离出来，那么使缓存/索引与主数据库保持同步通常是应用代码的责任。图1-1 给出了这种架构可能的样子（细节将在后面的章节中详细介绍）。

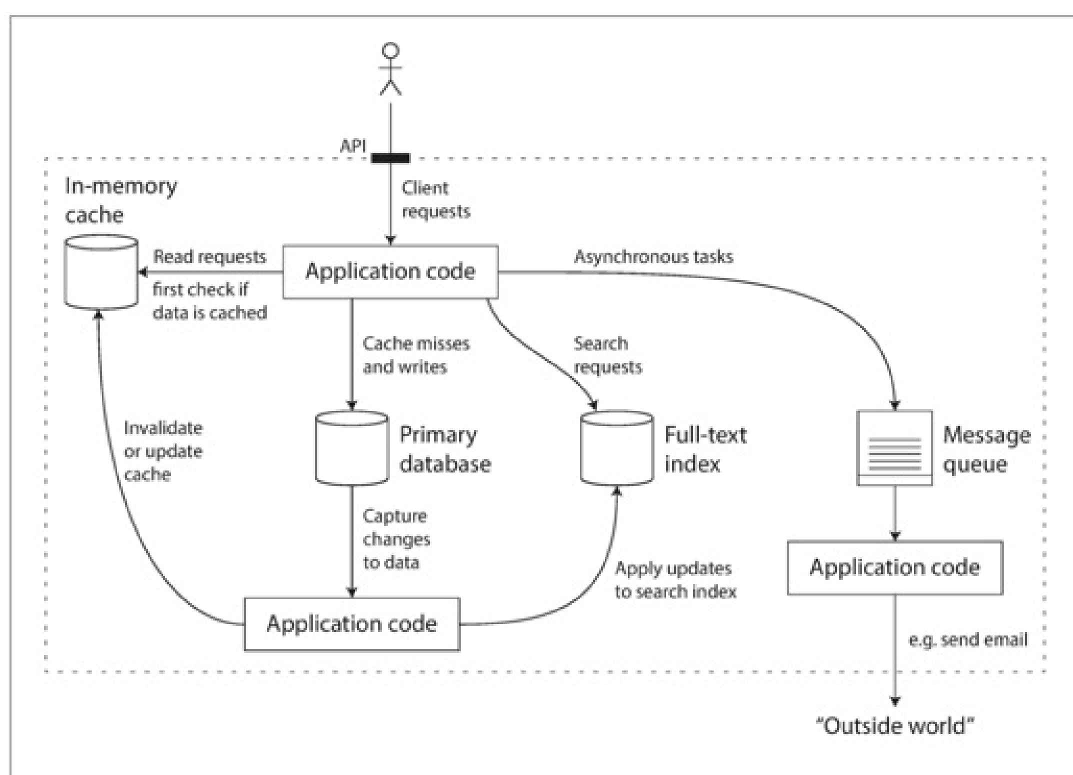


图**1-1** 一个可能的组合使用多个组件的数据系统架构

当你将多个工具组合在一起提供服务时，服务的接口或应用程序编程接口（**API, Application Programming Interface**）通常向客户端隐藏这些实现细节。现在，你基本上已经使用较小的通用组件创建了一个全新的、专用的数据系统。这个新的复合数据系统可能会提供特定的

保证，例如：缓存在写入时会作废或更新，以便外部客户端获取一致的结果。现在你不仅是应用程序开发人员，还是数据系统设计人员了。

设计数据系统或服务时可能会遇到很多棘手的问题，例如：当系统出问题时，如何确保数据的正确性和完整性？当部分系统退化降级时，如何为客户提供始终如一的良好性能？当负载增加时，如何扩容应对？什么样的API才是好的API？

影响数据系统设计的因素很多，包括参与人员的技能和经验、历史遗留问题、系统路径依赖、交付时限、公司的风险容忍度、监管约束等，这些因素都需要具体问题具体分析。

本书着重讨论三个在大多数软件系统中都很重要的问题：

可靠性（***Reliability***）

系统在困境（**adversity**）（硬件故障、软件故障、人为错误）中仍可正常工作（正确完成功能，并能达到期望的性能水准）。

可扩展性（***Scalability***）

有合理的办法应对系统的增长（数据量、流量、复杂性）（参阅“可扩展性”）

可维护性（***Maintainability***）

许多不同的人（工程师、运维）在不同的生命周期，都能在高效地在系统上工作（使系统保持现有行为，并适应新的应用场景）。（参阅”可维护性“）

人们经常追求这些词汇，却没有清楚理解它们到底意味着什么。为了工程的严谨性，本章的剩余部分将探讨可靠性、可扩展性、可维护性的含义。为实现这些目标而使用的各种技术，架构和算法将在后续的章节中研究。

可靠性

人们对于一个东西是否可靠，都有一个直观的想法。人们对可靠软件的典型期望包括：

- 应用程序表现出用户所期望的功能。
- 允许用户犯错，允许用户以出乎意料的方式使用软件。
- 在预期的负载和数据量下，性能满足要求。
- 系统能防止未经授权的访问和滥用。

如果所有这些在一起意味着“正确工作”，那么可以把可靠性粗略理解为“即使出现问题，也能继续正确工作”。

造成错误的原因叫做故障（**fault**），能预料并应对故障的系统特性可称为容错（**fault-tolerant**）或韧性（**resilient**）。“容错”一词可能会产生误导，因为它暗示着系统可以容忍所有可能的错误，但在实际中这是不可能的。比方说，如果整个地球（及其上的所有服务器）

都被黑洞吞噬了，想要容忍这种错误，需要把网络托管到太空中——这种预算能不能批准就祝你好运了。所以在讨论容错时，只有谈论特定类型的错误才有意义。

注意故障（**fault**）不同于失效（**failure**）【2】。故障通常定义为系统的一部分状态偏离其标准，而失效则是系统作为一个整体停止向用户提供服务。故障的概率不可能降到零，因此最好设计容错机制以防因故障而导致失效。本书们将介绍几种用不可靠的部件构建可靠系统的技术。

反直觉的是，在这类容错系统中，通过故意触发来提高故障率是有意义的，例如：在没有警告的情况下随机地杀死单个进程。许多高危漏洞实际上是由糟糕的错误处理导致的【3】，因此我们可以通过故意引发故障来确保容错机制不断运行并接受考验，从而提高故障自然发生时系统能正确处理的信心。Netflix公司的*Chaos Monkey*【4】就是这种方法的一个例子。

尽管比起阻止错误（**prevent error**），我们通常更倾向于容忍错误。但也有预防胜于治疗的情况（比如不存在治疗方法时）。安全问题就属于这种情况。例如，如果攻击者破坏了系统，并获取了敏感数据，这种事是撤销不了的。但本书主要讨论的是可以恢复的故障种类，正如下面几节所述。

硬件故障

当想到系统失效的原因时，硬件故障（**hardware faults**）总会第一个进入脑海。硬盘崩溃、内存出错、机房断电、有人拔错网线……任何与大型数据中心打过交道的人都会告诉你：一旦你拥有很多机器，这些事情总会发生！

据报道称，硬盘的平均无故障时间（**MTTF, mean time to failure**）约为10到50年【5】【6】。因此从数学期望上讲，在拥有10000个磁盘的存储集群上，平均每天会有1个磁盘出故障。

为了减少系统的故障率，第一反应通常都是增加单个硬件的冗余度，例如：磁盘可以组建RAID，服务器可能有双路电源和热插拔CPU，数据中心可能有电池和柴油发电机作为后备电源，某个组件挂掉时冗余组件可以立刻接管。这种方法虽然不能完全防止由硬件问题导致的系统失效，但它简单易懂，通常也足以让机器不间断运行很多年。

直到最近，硬件冗余对于大多数应用来说已经足够了，它使单台机器完全失效变得相当罕见。只要你能快速地把备份恢复到新机器上，故障停机时间对大多数应用而言都算不上灾难性的。只有少量高可用性至关重要的应用才会要求有多套硬件冗余。

但是随着数据量和应用计算需求的增加，越来越多的应用开始大量使用机器，这会相应地增加硬件故障率。此外在一些云平台（如亚马逊网络服务（**AWS, Amazon Web Services**））中，虚拟机实例不可用却没有任何警告也是很常见的【7】，因为云平台的设计就是优先考虑灵活性（**flexibility**）和弹性（**elasticity**）[i]，而不是单机可靠性。

如果在硬件冗余的基础上进一步引入软件容错机制，那么系统在容忍整个（单台）机器故障的道路上就更进一步了。这样的系统也有运维上的便利，例如：如果需要重启机器（例如应用操作系统安全补丁），单服务器系统就需要计划停机。而允许机器失效的系统则可以一次修复一个节点，无需整个系统停机。

> i. 在应对负载的方法一节定义 ↩

软件错误

我们通常认为硬件故障是随机的、相互独立的：一台机器的磁盘失效并不意味着另一台机器的磁盘也会失效。大量硬件组件不可能同时发生故障，除非它们存在比较弱的相关性（同样的原因导致关联性错误，例如服务器机架的温度）。

另一类错误是内部的系统性错误（**systematic error**）【7】。这类错误难以预料，而且因为是跨节点相关的，所以比起不相关的硬件故障往往可能造成更多的系统失效【5】。例子包括：

- 接受特定的错误输入，便导致所有应用服务器实例崩溃的BUG。例如2012年6月30日的闰秒，由于Linux内核中的一个错误，许多应用同时挂掉了。
- 失控进程会占用一些共享资源，包括CPU时间、内存、磁盘空间或网络带宽。
- 系统依赖的服务变慢，没有响应，或者开始返回错误的响应。
- 级联故障，一个组件中的小故障触发另一个组件中的故障，进而触发更多的故障【10】。

导致这类软件故障的BUG通常会潜伏很长时间，直到被异常情况触发为止。这种情况意味着软件对其环境做出了某种假设——虽然这种假设通常来说是正确的，但由于某种原因最后不再成立了【11】。

虽然软件中的系统性故障没有速效药，但我们还是有很多小办法，例如：仔细考虑系统中的假设和交互；彻底的测试；进程隔离；允许进程崩溃并重启；测量、监控并分析生产环境中的系统行为。如果系统能够提供一些保证（例如在一个消息队列中，进入与发出的消息数量相等），那么系统就可以在运行时不断自检，并在出现差异（**discrepancy**）时报警【12】。

人为错误

设计并构建了软件系统的工程师是人类，维持系统运行的运维也是人类。即使他们怀有最大的善意，人类也是不可靠的。举个例子，一项关于大型互联网服务的研究发现，运维配置错误是导致服务中断的首要原因，而硬件故障（服务器或网络）仅导致了10-25%的服务中断【13】。

尽管人类不可靠，但怎么做才能让系统变得可靠？最好的系统会组合使用以下几种办法：

- 以最小化犯错机会的方式设计系统。例如，精心设计的抽象、API和管理后台使做对事情

更容易，搞砸事情更困难。但如果接口限制太多，人们就会忽略它们的好处而想办法绕开。很难正确把握这种微妙的平衡。

* 将人们最容易犯错的地方与可能导致失效的地方解耦（**decouple**）。特别是提供一个功能齐全的非生产环境沙箱（**sandbox**），使人们可以在不影响真实用户的情况下，使用真实数据安全地探索和实验。
* 在各个层次进行彻底的测试【3】，从单元测试、全系统集成测试到手动测试。自动化测试易于理解，已经被广泛使用，特别适合用来覆盖正常情况中少见的边缘场景（**corner case**）。
* 允许从人为错误中简单快速地恢复，以最大限度地减少失效情况带来的影响。 例如，快速回滚配置变更，分批发布新代码（以便任何意外错误只影响一小部分用户），并提供数据重算工具（以备旧的计算出错）。
* 配置详细和明确的监控，比如性能指标和错误率。 在其他工程学科中这指的是遥测（**telemetry**）。 （一旦火箭离开了地面，遥测技术对于跟踪发生的事情和理解失败是至关重要的。）监控可以向我们发出预警信号，并允许我们检查是否有任何地方违反了假设和约束。当出现问题时，指标数据对于问题诊断是非常宝贵的。
* 良好的管理实践与充分的培训——一个复杂而重要的方面，但超出了本书的范围。

可靠性有多重要？

可靠性不仅仅是针对核电站和空中交通管制软件而言，我们也期望更多平凡的应用能可靠地运行。商务应用中的错误会导致生产力损失（也许数据报告不完整还会有法律风险），而电商网站的中断则可能会导致收入和声誉的巨大损失。

即使在“非关键”应用中，我们也对用户负有责任。试想一位家长把所有的照片和孩子的视频储存在你的照片应用里【15】。如果数据库突然损坏，他们会感觉如何？他们可能会知道如何从备份恢复吗？

在某些情况下，我们可能会选择牺牲可靠性来降低开发成本（例如为未经证实的市场开发产品原型）或运营成本（例如利润率极低的服务），但我们偷工减料时，应该清楚意识到自己在做什么。

可扩展性

系统今天能可靠运行，并不意味未来也能可靠运行。服务降级（**degradation**）的一个常见原因是负载增加，例如：系统负载已经从一万个并发用户增长到十万个并发用户，或者从一百万增长到一千万。也许现在处理的数据量级要比过去大得多。

可扩展性（**Scalability**）是用来描述系统应对负载增长能力的术语。但是请注意，这不是贴在系统上的一维标签：说“X可扩展”或“Y不可扩展”是没有任何意义的。相反，讨论可扩展性意味着考虑诸如“如果系统以特定方式增长，有什么选项可以应对增长？”和“如何增加计算资源来处理额外的负载？”等问题。

描述负载

在讨论增长问题（如果负载加倍会发生什么？）前，首先要能简要描述系统的当前负载。负载可以用一些称为负载参数（**load parameters**）的数字来描述。参数的最佳选择取决于系统架构，它可能是每秒向Web服务器发出的请求、数据库中的读写比率、聊天室中同时活跃的用户数量、缓存命中率或其他东西。除此之外，也许平均情况对你很重要，也许你的瓶颈是少数极端场景。

为了使这个概念更加具体，我们以推特在2012年11月发布的数据【16】为例。推特的两个主要业务是：

发布推文

用户可以向其粉丝发布新消息（平均 4.6k请求/秒，峰值超过 12k请求/秒）。

主页时间线

用户可以查阅他们关注的人发布的推文（300k请求/秒）。

处理每秒12,000次写入（发推文的速率峰值）还是很简单的。然而推特的扩展性挑战并不是主要来自推特量，而是来自扇出（**fan-out**）——每个用户关注了很多人，也被很多人关注。

> ii. 扇出：从电子工程学中借用的术语，它描述了输入连接到另一个门输出的逻辑门数量。 输出需要提供足够的电流来驱动所有连接的输入。 在事务处理系统中，我们使用它来描述为了服务一个传入请求而需要执行其他服务的请求数量。 ↩

大体上讲，这一对操作有两种实现方式。

1. 发布推文时，只需将新推文插入全局推文集合即可。当一个用户请求自己的主页时间线时，首先查找他关注的所有人，查询这些被关注用户发布的推文并按时间顺序合并。在如图1-2所示的关系型数据库中，可以编写这样的查询：

```
SELECT tweets.*, users.*
  FROM tweets
  JOIN users   ON tweets.sender_id = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

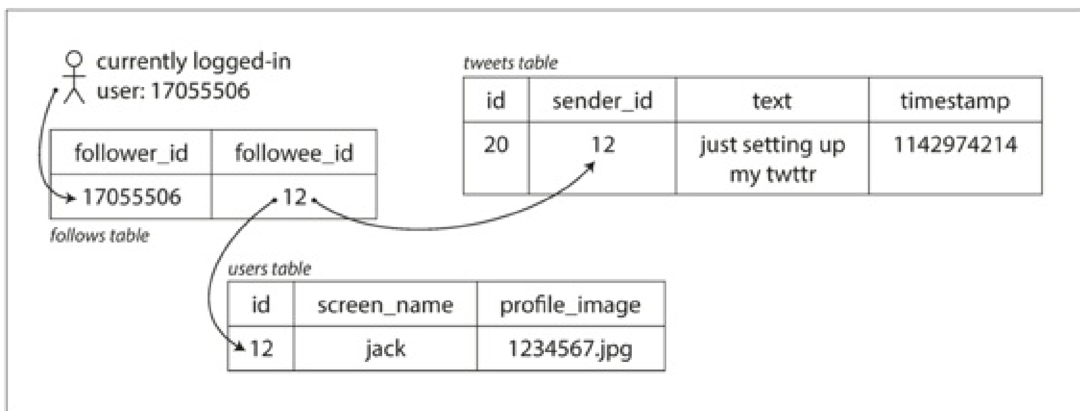


图**1-2** 推特主页时间线的关系型模式简单实现

2. 为每个用户的主页时间线维护一个缓存，就像每个用户的推文收件箱（图1-3）。 当一个用户发布推文时，查找所有关注该用户的人，并将新的推文插入到每个主页时间线缓存中。 因此读取主页时间线的请求开销很小，因为结果已经提前计算好了。

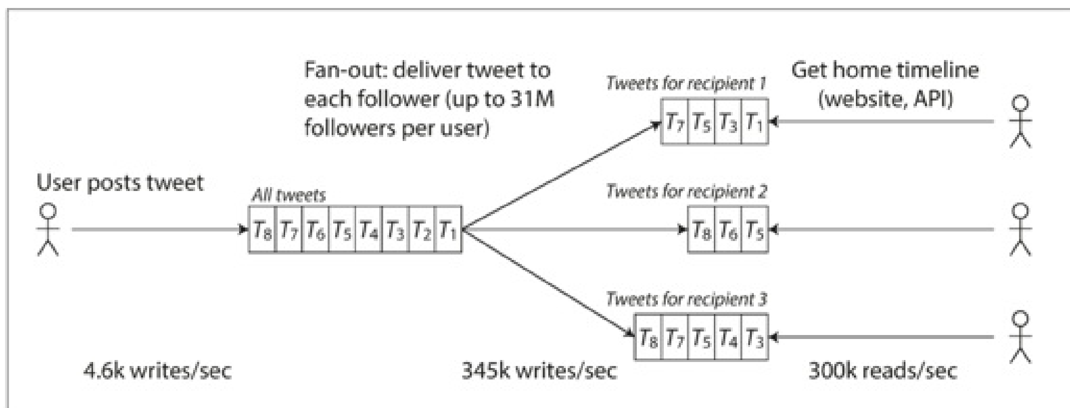


图**1-3** 用于分发推特至关注者的数据流水线，**2012**年**11**月的负载参数【**16**】

推特的第一个版本使用了方法1，但系统很难跟上主页时间线查询的负载。所以公司转向了方法2，方法2的效果更好，因为发推频率比查询主页时间线的频率几乎低了两个数量级，所以在这种情况下，最好在写入时做更多的工作，而在读取时做更少的工作。

然而方法2的缺点是，发推现在需要大量的额外工作。平均来说，一条推文会发往约75个关注者，所以每秒4.6k的发推写入，变成了对主页时间线缓存每秒345k的写入。但这个平均值隐藏了用户粉丝数差异巨大这一现实，一些用户有超过3000万的粉丝，这意味着一条推文就可能会导致主页时间线缓存的3000万次写入！及时完成这种操作是一个巨大的挑战 —— 推特尝试在5秒内向粉丝发送推文。

在推特的例子中，每个用户粉丝数的分布（可能按这些用户的发推频率来加权）是探讨可扩展性的一个关键负载参数，因为它决定了扇出负载。你的应用程序可能具有非常不同的特征，但可以采用相似的原则来考虑它的负载。

推特轶事的最终转折：现在已经稳健地实现了方法2，推特逐步转向了两种方法的混合。大多数用户发的推文会被扇出写入其粉丝主页时间线缓存中。但是少数拥有海量粉丝的用户（即名流）会被排除在外。当用户读取主页时间线时，分别地获取出该用户所关注的每位名流的推文，再与用户的主页时间线缓存合并，如方法1所示。这种混合方法能始终如一地提供良好性能。在第12章中我们将重新讨论这个例子，这在覆盖更多技术层面之后。

描述性能

一旦系统的负载被描述好，就可以研究当负载增加会发生什么。我们可以从两种角度来看：

- 增加负载参数并保持系统资源（CPU、内存、网络带宽等）不变时，系统性能将受到什么影响？
- 增加负载参数并希望保持性能不变时，需要增加多少系统资源？

这两个问题都需要性能数据，所以让我们简单地看一下如何描述系统性能。

对于Hadoop这样的批处理系统，通常关心的是吞吐量（**throughput**），即每秒可以处理的记录数量，或者在特定规模数据集上运行作业的总时间[iii]。对于在线系统，通常更重要的是服务的响应时间（**response time**），即客户端发送请求到接收响应之间的时间。

> iii. 理想情况下，批量作业的运行时间是数据集的大小除以吞吐量。 在实践中由于数据倾斜（数据不是均匀分布在每个工作进程中），需要等待最慢的任务完成，所以运行时间往往更长。↩

> 延迟和响应时间
>
> 延迟（**latency**）和响应时间（**response time**）经常用作同义词，但实际上它们并不一样。响应时间是客户所看到的，除了实际处理请求的时间（服务时间（**service time**））之外，还包括网络延迟和排队延迟。延迟是某个请求等待处理的持续时长，在此期间它处于休眠（**latent**）状态，并等待服务【17】。

即使不断重复发送同样的请求，每次得到的响应时间也都会略有不同。现实世界的系统会处理各式各样的请求，响应时间可能会有很大差异。因此我们需要将响应时间视为一个可以测量的数值分布（**distribution**），而不是单个数值。

在图1-4中，每个灰条表代表一次对服务的请求，其高度表示请求花费了多长时间。大多数请求是相当快的，但偶尔会出现需要更长的时间的异常值。这也许是因为缓慢的请求实质上开销更大，例如它们可能会处理更多的数据。但即使（你认为）所有请求都花费相同时间的情况下，随机的附加延迟也会导致结果变化，例如：上下文切换到后台进程，网络数据包丢失与TCP重传，垃圾收集暂停，强制从磁盘读取的页面错误，服务器机架中的震动【18】，还有很多其他原因。

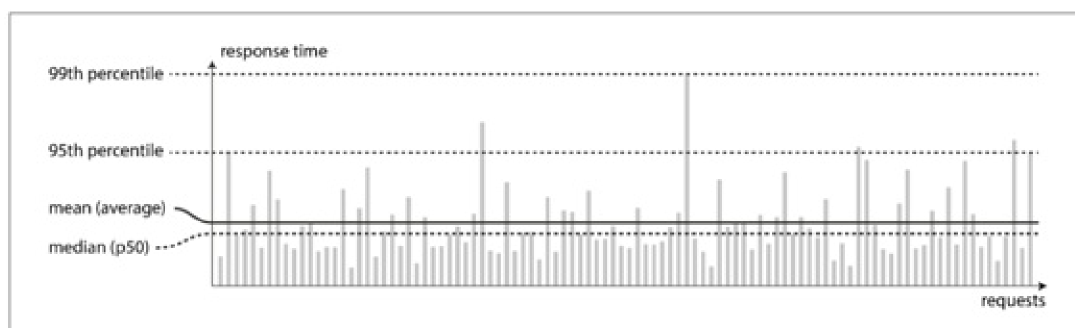


图**1-4** 展示了一个服务**100**次请求响应时间的均值与百分位数

通常报表都会展示服务的平均响应时间。 （严格来讲“平均”一词并不指代任何特定公式，但实际上它通常被理解为算术平均值（**arithmetic mean**）：给定 n 个值，加起来除以 n ）。然而如果你想知道“典型（**typical**）”响应时间，那么平均值并不是一个非常好的指标，因为它不能告诉你有多少用户实际上经历了这个延迟。

通常使用百分位点（**percentiles**）会更好。如果将响应时间列表按最快到最慢排序，那么中位数（**median**）就在正中间：举个例子，如果你的响应时间中位数是200毫秒，这意味着一半请求的返回时间少于200毫秒，另一半比这个要长。

如果想知道典型场景下用户需要等待多长时间，那么中位数是一个好的度量标准：一半用户请求的响应时间少于响应时间的中位数，另一半服务时间比中位数长。中位数也被称为第50百分位点，有时缩写为p50。注意中位数是关于单个请求的；如果用户同时发出几个请求（在一个会话过程中，或者由于一个页面中包含了多个资源），则至少一个请求比中位数慢的概率远大于50%。

为了弄清异常值有多糟糕，可以看看更高的百分位点，例如第95、99和99.9百分位点（缩写为p95，p99和p999）。它们意味着95%，99%或99.9%的请求响应时间要比该阈值快，例如：如果第95百分位点响应时间是1.5秒，则意味着100个请求中的95个响应时间快于1.5秒，而100个请求中的5个响应时间超过1.5秒。如图1-4所示。

响应时间的高百分位点（也称为尾部延迟（**tail latencies**））非常重要，因为它们直接影响用户的服务体验。例如亚马逊在描述内部服务的响应时间要求时以99.9百分位点为准，即使它只影响一千个请求中的一个。这是因为请求响应最慢的客户往往也是数据最多的客户，也可以说是最有价值的客户 —— 因为他们掏钱了【19】。保证网站响应迅速对于保持客户的满意度非常重要，亚马逊观察到：响应时间增加100毫秒，销售量就减少1%【20】；而另一些报告说：慢 1 秒钟会让客户满意度指标减少16%【21，22】。

另一方面，优化第99.99百分位点（一万个请求中最慢的一个）被认为太昂贵了，不能为亚马逊的目标带来足够好处。减小高百分位点处的响应时间相当困难，因为它很容易受到随机事件的影响，这超出了控制范围，而且效益也很小。

百分位点通常用于服务级别目标（**SLO, service level objectives**）和服务级别协议（**SLA, service level agreements**），即定义服务预期性能和可用性的合同。 SLA可能会声明，如果服务响应时间的中位数小于200毫秒，且99.9百分位点低于1秒，则认为服务工作正常（如果响应时间更长，就认为服务不达标）。这些指标为客户设定了期望值，并允许客户在SLA未达标的情况下要求退款。

排队延迟（**queueing delay**）通常占了高百分位点处响应时间的很大一部分。由于服务器只能并行处理少量的事务（如受其CPU核数的限制），所以只要有少量缓慢的请求就能阻碍后续请求的处理，这种效应有时被称为头部阻塞（**head-of-line blocking**）。即使后续请求在服务器上处理的非常迅速，由于需要等待先前请求完成，客户端最终看到的是缓慢的总体响应时间。因为存在这种效应，测量客户端的响应时间非常重要。

为测试系统的可扩展性而人为产生负载时，产生负载的客户端要独立于响应时间不断发送请求。如果客户端在发送下一个请求之前等待先前的请求完成，这种行为会产生人为排队的效果，使得测试时的队列比现实情况更短，使测量结果产生偏差【23】。

> 实践中的百分位点
>
> 在多重调用的后端服务里，高百分位数变得特别重要。即使并行调用，最终用户请求仍然需要等待最慢的并行呼叫完成。如图1-5所示，只需要一个缓慢的呼叫就可以使整个最终用户请求变慢。即使只有一小部分后端呼叫速度较慢，如果最终用户请求需要多个后端调用，则获得较慢调用的机会也会增加，因此较高比例的最终用户请求速度会变慢（效果称为尾部延迟放大【24】）。
>
> 如果您想将响应时间百分点添加到您的服务的监视仪表板，则需要持续有效地计算它们。例如，您可能希望在最近10分钟内保持请求响应时间的滚动窗口。每一分钟，您都会计算出该窗口中的中值和各种百分数，并将这些度量值绘制在图上。
>
> 简单的实现是在时间窗口内保存所有请求的响应时间列表，并且每分钟对列表进行排序。如果对你来说效率太低，那么有一些算法能够以最小的CPU和内存成本（如前向衰减【25】，t-digest【26】或HdrHistogram 【27】）来计算百分位数的近似值。请注意，平均百分比（例如，减少时间分辨率或合并来自多台机器的数据）在数学上没有意义 - 聚合响应时间数据的正确方法是添加直方图【28】。

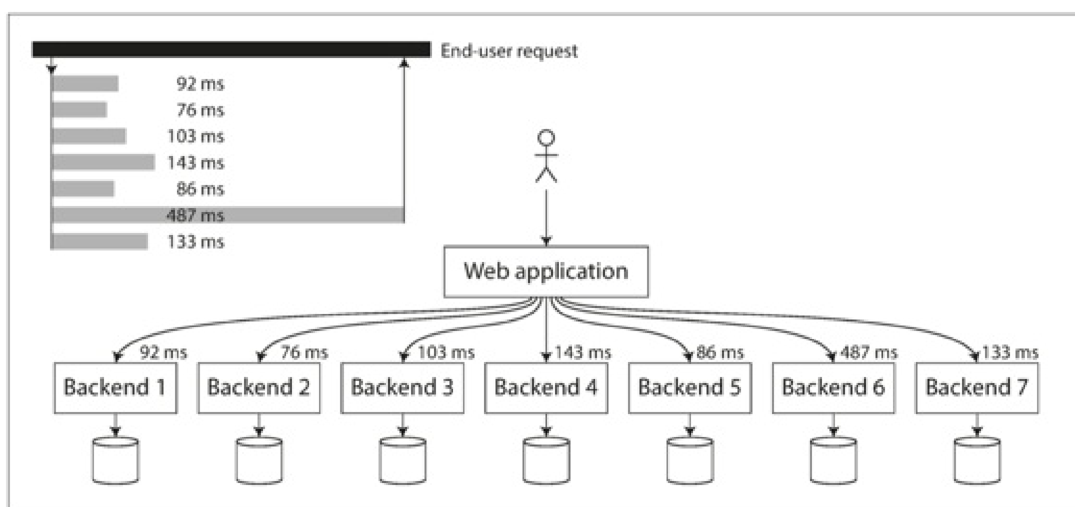


图**1-5** 当一个请求需要多个后端请求时，单个后端慢请求就会拖慢整个终端用户的请求

应对负载的方法

现在我们已经讨论了用于描述负载的参数和用于衡量性能的指标。可以开始认真讨论可扩展性了：当负载参数增加时，如何保持良好的性能？

适应某个级别负载的架构不太可能应付10倍于此的负载。如果你正在开发一个快速增长的服务，那么每次负载发生数量级的增长时，你可能都需要重新考虑架构——或者更频繁。

人们经常讨论纵向扩展（**scaling up**）（垂直扩展（**vertical scaling**），转向更强大的机器）和横向扩展（**scaling out**）（水平扩展（**horizontal scaling**），将负载分布到多台小机器上）之间的对立。跨多台机器分配负载也称为"无共享（**shared-nothing**）"架构。可以在单

台机器上运行的系统通常更简单，但高端机器可能非常贵，所以非常密集的负载通常无法避免地需要横向扩展。现实世界中的优秀架构需要将这两种方法务实地结合，因为使用几台足够强大的机器可能比使用大量的小型虚拟机更简单也更便宜。

有些系统是弹性（**elastic**）的，这意味着可以在检测到负载增加时自动增加计算资源，而其他系统则是手动扩展（人工分析容量并决定向系统添加更多的机器）。如果负载极难预测（**highly unpredictable**），则弹性系统可能很有用，但手动扩展系统更简单，并且意外操作可能会更少（参阅“重新平衡分区”）。

跨多台机器部署无状态服务（**stateless services**）非常简单，但将带状态的数据系统从单节点变为分布式配置则可能引入许多额外复杂度。出于这个原因，常识告诉我们应该将数据库放在单个节点上（纵向扩展），直到扩展成本或可用性需求迫使其改为分布式。

随着分布式系统的工具和抽象越来越好，至少对于某些类型的应用而言，这种常识可能会改变。可以预见分布式数据系统将成为未来的默认设置，即使对不处理大量数据或流量的场景也如此。本书的其余部分将介绍多种分布式数据系统，不仅讨论它们在可扩展性方面的表现，还包括易用性和可维护性。

大规模的系统架构通常是应用特定的—— 没有一招鲜吃遍天的通用可扩展架构（不正式的叫法：万金油（**magic scaling sauce**））。应用的问题可能是读取量、写入量、要存储的数据量、数据的复杂度、响应时间要求、访问模式或者所有问题的大杂烩。

举个例子，用于处理每秒十万个请求（每个大小为1 kB）的系统与用于处理每分钟3个请求（每个大小为2GB）的系统看上去会非常不一样，尽管两个系统有同样的数据吞吐量。

一个良好适配应用的可扩展架构，是围绕着假设（**assumption**）建立的：哪些操作是常见的？哪些操作是罕见的？这就是所谓负载参数。如果假设最终是错误的，那么为扩展所做的工程投入就白费了，最糟糕的是适得其反。在早期创业公司或非正式产品中，通常支持产品快速迭代的能力，要比可扩展至未来的假想负载要重要的多。

尽管这些架构是应用程序特定的，但可扩展的架构通常也是从通用的积木块搭建而成的，并以常见的模式排列。在本书中，我们将讨论这些构件和模式。

可维护性

众所周知，软件的大部分开销并不在最初的开发阶段，而是在持续的维护阶段，包括修复漏洞、保持系统正常运行、调查失效、适配新的平台、为新的场景进行修改、偿还技术债、添加新的功能等等。

不幸的是，许多从事软件系统行业的人不喜欢维护所谓的遗留（**legacy**）系统，——也许因为涉及修复其他人的错误、和过时的平台打交道，或者系统被迫使用于一些份外工作。每一个遗留系统都以自己的方式让人不爽，所以很难给出一个通用的建议来和它们打交道。

但是我们可以，也应该以这样一种方式来设计软件：在设计之初就尽量考虑尽可能减少维护期间的痛苦，从而避免自己的软件系统变成遗留系统。为此，我们将特别关注软件系统的三个设计原则：

可操作性（***Operability***）

便于运维团队保持系统平稳运行。

简单性（***Simplicity***）

从系统中消除尽可能多的复杂度（**complexity**），使新工程师也能轻松理解系统。（注意这和用户接口的简单性不一样。）

可演化性（***evolability***）

使工程师在未来能轻松地对系统进行更改，当需求变化时为新应用场景做适配。也称为可扩展性（**extensibility**），可修改性（**modifiability**）或可塑性（**plasticity**）。

和之前提到的可靠性、可扩展性一样，实现这些目标也没有简单的解决方案。不过我们会试着想象具有可操作性，简单性和可演化性的系统会是什么样子。

可操作性：人生苦短，关爱运维

有人认为，"良好的运维经常可以绕开垃圾（或不完整）软件的局限性，而再好的软件摊上垃圾运维也没法可靠运行"。尽管运维的某些方面可以，而且应该是自动化的，但在最初建立正确运作的自动化机制仍然取决于人。

运维团队对于保持软件系统顺利运行至关重要。一个优秀运维团队的典型职责如下（或者更多）【29】：

- 监控系统的运行状况，并在服务状态不佳时快速恢复服务
- 跟踪问题的原因，例如系统故障或性能下降
- 及时更新软件和平台，比如安全补丁
- 了解系统间的相互作用，以便在异常变更造成损失前进行规避。
- 预测未来的问题，并在问题出现之前加以解决（例如，容量规划）
- 建立部署，配置、管理方面的良好实践，编写相应工具
- 执行复杂的维护任务，例如将应用程序从一个平台迁移到另一个平台
- 当配置变更时，维持系统的安全性
- 定义工作流程，使运维操作可预测，并保持生产环境稳定。
- 铁打的营盘流水的兵，维持组织对系统的了解。

良好的可操作性意味着更轻松的日常工作，进而运维团队能专注于高价值的事情。数据系统可以通过各种方式使日常任务更轻松：

- 通过良好的监控，提供对系统内部状态和运行时行为的可见性（**visibility**）
- 为自动化提供良好支持，将系统与标准化工具相集成

- 避免依赖单台机器（在整个系统继续不间断运行的情况下允许机器停机维护）
- 提供良好的文档和易于理解的操作模型（“如果做X，会发生Y”）
- 提供良好的默认行为，但需要时也允许管理员自由覆盖默认值
- 有条件时进行自我修复，但需要时也允许管理员手动控制系统状态
- 行为可预测，最大限度减少意外

简单性：管理复杂度

小型软件项目可以使用简单讨喜的、富表现力的代码，但随着项目越来越大，代码往往变得非常复杂，难以理解。这种复杂度拖慢了所有系统相关人员，进一步增加了维护成本。一个陷入复杂泥潭的软件项目有时被描述为烂泥潭（**a big ball of mud**）【30】。

复杂度（**complexity**）有各种可能的症状，例如：状态空间激增、模块间紧密耦合、纠结的依赖关系、不一致的命名和术语、解决性能问题的Hack、需要绕开的特例等等，现在已经有很多关于这个话题的讨论【31,32,33】。

因为复杂度导致维护困难时，预算和时间安排通常会超支。在复杂的软件中进行变更，引入错误的风险也更大：当开发人员难以理解系统时，隐藏的假设、无意的后果和意外的交互就更容易被忽略。相反，降低复杂度能极大地提高软件的可维护性，因此简单性应该是构建系统的一个关键目标。

简化系统并不一定意味着减少功能；它也可以意味着消除额外的（**accidental**）的复杂度。Moseley和Marks【32】把额外复杂度定义为：由具体实现中涌现，而非（从用户视角看，系统所解决的）问题本身固有的复杂度。

用于消除额外复杂度的最好工具之一是抽象（**abstraction**）。一个好的抽象可以将大量实现细节隐藏在一个干净，简单易懂的外观下面。一个好的抽象也可以广泛用于各类不同应用。比起重复造很多轮子，重用抽象不仅更有效率，而且有助于开发高质量的软件。抽象组件的质量改进将使所有使用它的应用受益。

例如，高级编程语言是一种抽象，隐藏了机器码、CPU寄存器和系统调用。 SQL也是一种抽象，隐藏了复杂的磁盘/内存数据结构、来自其他客户端的并发请求、崩溃后的不一致性。当然在用高级语言编程时，我们仍然用到了机器码；只不过没有直接（**directly**）使用罢了，正是因为编程语言的抽象，我们才不必去考虑这些实现细节。

抽象可以帮助我们将系统的复杂度控制在可管理的水平，不过，找到好的抽象是非常困难的。在分布式系统领域虽然有许多好的算法，但我们并不清楚它们应该打包成什么样抽象。

本书将紧盯那些允许我们将大型系统的部分提取为定义明确的、可重用的组件的优秀抽象。

可演化性：拥抱变化

系统的需求永远不变，基本是不可能的。更可能的情况是，它们处于常态的变化中，例如：你了解了新的事实、出现意想不到的应用场景、业务优先级发生变化、用户要求新功能、新平台取代旧平台、法律或监管要求发生变化、系统增长迫使架构变化等。

在组织流程方面，敏捷（**agile**）工作模式为适应变化提供了一个框架。敏捷社区还开发了对在频繁变化的环境中开发软件很有帮助的技术工具和模式，如测试驱动开发（**TDD, test-driven development**）和重构（**refactoring**）。

这些敏捷技术的大部分讨论都集中在相当小的规模（同一个应用中的几个代码文件）。本书将探索在更大数据系统层面上提高敏捷性的方法，可能由几个不同的应用或服务组成。例如，为了将装配主页时间线的方法从方法1变为方法2，你会如何“重构”推特的架构 ？

修改数据系统并使其适应不断变化需求的容易程度，是与简单性和抽象性密切相关的：简单易懂的系统通常比复杂系统更容易修改。但由于这是一个非常重要的概念，我们将用一个不同的词来指代数据系统层面的敏捷性：可演化性（**evolvability**）【34】。

本章小结

本章探讨了一些关于数据密集型应用的基本思考方式。这些原则将指导我们阅读本书的其余部分，那里将会深入技术细节。

一个应用必须满足各种需求才称得上有用。有一些功能需求（**functional requirements**）（它应该做什么，比如允许以各种方式存储，检索，搜索和处理数据）以及一些非功能性需求（**nonfunctional** ）（通用属性，例如安全性，可靠性，合规性，可扩展性，兼容性和可维护性）。在本章详细讨论了可靠性，可扩展性和可维护性。

可靠性（**Reliability**）意味着即使发生故障，系统也能正常工作。故障可能发生在硬件（通常是随机的和不相关的），软件（通常是系统性的Bug，很难处理），和人类（不可避免地时不时出错）。容错技术可以对终端用户隐藏某些类型的故障。

可扩展性（**Scalability**）意味着即使在负载增加的情况下也有保持性能的策略。为了讨论可扩展性，我们首先需要定量描述负载和性能的方法。我们简要了解了推特主页时间线的例子，介绍描述负载的方法，并将响应时间百分位点作为衡量性能的一种方式。在可扩展的系统中可以添加处理容量（**processing capacity**）以在高负载下保持可靠。

可维护性（**Maintainability**）有许多方面，但实质上是关于工程师和运维团队的生活质量的。良好的抽象可以帮助降低复杂度，并使系统易于修改和适应新的应用场景。良好的可操作性意味着对系统的健康状态具有良好的可见性，并拥有有效的管理手段。

不幸的是，使应用可靠、可扩展或可维护并不容易。但是某些模式和技术会不断重新出现在不同的应用中。在接下来的几章中，我们将看到一些数据系统的例子，并分析它们如何实现这些目标。

在本书后面的第三部分中，我们将看到一种模式：几个组件协同工作以构成一个完整的系统（如图1-1中的例子）

参考文献

1. Michael Stonebraker and Uğur Çetintemel: “'One Size Fits All': An Idea Whose Time Has Come and Gone,” at *21st International Conference on Data Engineering* (ICDE), April 2005.
2. Walter L. Heimerdinger and Charles B. Weinstock: “A Conceptual Framework for System Fault Tolerance,” Technical Report CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, October 1992.
3. Ding Yuan, Yu Luo, Xin Zhuang, et al.: “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems,” at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.
4. Yury Izrailevsky and Ariel Tseitlin: “The Netflix Simian Army,” *techblog.netflix.com*, July 19, 2011.
5. Daniel Ford, François Labelle, Florentina I. Popovici, et al.: “Availability in Globally Distributed Storage Systems,” at *9th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2010.
6. Brian Beach: “Hard Drive Reliability Update – Sep 2014,” *backblaze.com*, September 23, 2014.
7. Laurie Voss: “AWS: The Good, the Bad and the Ugly,” *blog.awe.sm*, December 18, 2012.
8. Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: “What Bugs Live in the Cloud?,” at *5th ACM Symposium on Cloud Computing* (SoCC), November 2014. doi:10.1145/2670979.2670986
9. Nelson Minar: “Leap Second Crashes Half the Internet,” *somebits.com*, July 3, 2012.
10. Amazon Web Services: “Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region,” *aws.amazon.com*, April 29, 2011.
11. Richard I. Cook: “How Complex Systems Fail,” Cognitive Technologies Laboratory, April 2000.
12. Jay Kreps: “Getting Real About Distributed System Reliability,” *blog.empathybox.com*, March 19, 2012.

13. David Oppenheimer, Archana Ganapathi, and David A. Patterson: “Why Do Internet Services Fail, and What Can Be Done About It?,” at *4th USENIX Symposium on Internet Technologies and Systems* (USITS), March 2003.
14. Nathan Marz: “Principles of Software Engineering, Part 1,” *nathanmarz.com*, April 2, 2013.
15. Michael Jurewitz:“The Human Impact of Bugs,” *jury.me*, March 15, 2013.
16. Raffi Krikorian: “Timelines at Scale,” at *QCon San Francisco*, November 2012.
17. Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. ISBN: 978-0-321-12742-6
18. Kelly Sommers: “After all that run around, what caused 500ms disk latency even when we replaced physical server?” *twitter.com*, November 13, 2014.
19. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: “Dynamo: Amazon's Highly Available Key-Value Store,” at *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007.
20. Greg Linden: “Make Data Useful,” slides from presentation at Stanford University Data Mining class (CS345), December 2006.
21. Tammy Everts: “The Real Cost of Slow Time vs Downtime,” *webperformancetoday.com*, November 12, 2014.
22. Jake Brutlag:“Speed Matters for Google Web Search,” *googleresearch.blogspot.co.uk*, June 22, 2009.
23. Tyler Treat: “Everything You Know About Latency Is Wrong,” *bravenewgeek.com*, December 12, 2015.
24. Jeffrey Dean and Luiz André Barroso: “The Tail at Scale,” *Communications of the ACM*, volume 56, number 2, pages 74–80, February 2013. doi:10.1145/2408776.2408794
25. Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu: “Forward Decay: A Practical Time Decay Model for Streaming Systems,” at *25th IEEE International Conference on Data Engineering* (ICDE), March 2009.
26. Ted Dunning and Otmar Ertl: “Computing Extremely Accurate Quantiles Using t-Digests,” *github.com*, March 2014.
27. Gil Tene: “HdrHistogram,” *hdrhistogram.org*.
28. Baron Schwartz: “Why Percentiles Don’t Work the Way You Think,” *vividcortex.com*, December 7, 2015.

29. James Hamilton: "On Designing and Deploying Internet-Scale Services," at *21st Large Installation System Administration Conference* (LISA), November 2007.
30. Brian Foote and Joseph Yoder: "Big Ball of Mud," at *4th Conference on Pattern Languages of Programs* (PLoP), September 1997.
31. Frederick P Brooks: "No Silver Bullet – Essence and Accident in Software Engineering," in *The Mythical Man-Month*, Anniversary edition, Addison-Wesley, 1995. ISBN: 978-0-201-83595-3
32. Ben Moseley and Peter Marks: "Out of the Tar Pit," at *BCS Software Practice Advancement* (SPA), 2006.
33. Rich Hickey: "Simple Made Easy," at *Strange Loop*, September 2011.
34. Hongyu Pei Breivold, Ivica Crnkovic, and Peter J. Eriksson: "Analyzing Software Evolvability," at *32nd Annual IEEE International Computer Software and Applications Conference* (COMPSAC), July 2008. doi:10.1109/COMPSAC.2008.50

上一章	目录	下一章
第一部分：数据系统基础	设计数据密集型应用	第二章：数据模型与查询语言

2. 数据模型与查询语言



> 语言的边界就是思想的边界。
>
> —— 路德维奇·维特根斯坦，《逻辑哲学》（1922）

[TOC]

数据模型可能是软件开发中最重要的部分了，因为它们的影响如此深远：不仅仅影响着软件的编写方式，而且影响着我们的解题思路。

多数应用使用层层叠加的数据模型构建。对于每层数据模型的关键问题是：它是如何用低一层数据模型来表示的？例如：

1. 作为一名应用开发人员，你观察现实世界（里面有人员，组织，货物，行为，资金流向，传感器等），并采用对象或数据结构，以及操控那些数据结构的API来进行建模。那些结构通常是特定于应用程序的。
2. 当要存储那些数据结构时，你可以利用通用数据模型来表示它们，如JSON或XML文档，关系数据库中的表、或图模型。
3. 数据库软件的工程师选定如何以内存、磁盘或网络上的字节来表示JSON/XML/关系/图数据。这类表示形式使数据有可能以各种方式来查询，搜索，操纵和处理。
4. 在更低的层次上，硬件工程师已经想出了使用电流，光脉冲，磁场或者其他东西来表示字节的方法。

一个复杂的应用程序可能会有更多的中间层次，比如基于API的API，不过基本思想仍然是一样的：每个层都通过提供一个明确的数据模型来隐藏更低层次中的复杂性。这些抽象允许不同的人群有效地协作（例如数据库厂商的工程师和使用数据库的应用程序开发人员）。

数据模型种类繁多，每个数据模型都带有如何使用的设想。有些用法很容易，有些则不支持如此；有些操作运行很快，有些则表现很差；有些数据转换非常自然，有些则很麻烦。

掌握一个数据模型需要花费很多精力（想想关系数据建模有多少本书）。即便只使用一个数据模型，不用操心其内部工作机制，构建软件也是非常困难的。然而，因为数据模型对上层软件的功能（能做什么，不能做什么）有着至深的影响，所以选择一个适合的数据模型是非常重要的。

在本章中，我们将研究一系列用于数据存储和查询的通用数据模型（前面列表中的第2点）。特别地，我们将比较关系模型，文档模型和少量基于图形的数据模型。我们还将查看各种查询语言并比较它们的用例。在第3章中，我们将讨论存储引擎是如何工作的。也就是说，这些数据模型实际上是如何实现的（列表中的第3点）。

关系模型与文档模型

现在最著名的数据模型可能是SQL。它基于Edgar Codd在1970年提出的关系模型【1】：数据被组织成关系（SQL中称作表），其中每个关系是元组（SQL中称作行)的无序集合。

关系模型曾是一个理论性的提议，当时很多人都怀疑是否能够有效实现它。然而到了20世纪80年代中期，关系数据库管理系统（RDBMSes）和SQL已成为大多数人们存储和查询某些常规结构的数据的首选工具。关系数据库已经持续称霸了大约25~30年——这对计算机史来说是极其漫长的时间。

关系数据库起源于商业数据处理，在20世纪60年代和70年代用大型计算机来执行。从今天的角度来看，那些用例显得很平常：典型的事务处理（将销售或银行交易，航空公司预订，库存管理信息记录在库）和批处理（客户发票，工资单，报告）。

当时的其他数据库迫使应用程序开发人员必须考虑数据库内部的数据表示形式。关系模型致力于将上述实现细节隐藏在更简洁的接口之后。

多年来，在数据存储和查询方面存在着许多相互竞争的方法。在20世纪70年代和80年代初，网络模型和分层模型曾是主要的选择，但关系模型随后占据了主导地位。对象数据库在20世纪80年代末和90年代初来了又去。XML数据库在二十一世纪初出现，但只有小众采用过。关系模型的每个竞争者都在其时代产生了大量的炒作，但从来没有持续【2】。

随着电脑越来越强大和互联，它们开始用于日益多样化的目的。关系数据库非常成功地被推广到业务数据处理的原始范围之外更为广泛的用例上。你今天在网上看到的大部分内容依旧是由关系数据库来提供支持，无论是在线发布，讨论，社交网络，电子商务，游戏，软件即服务生产力应用程序等等内容。

NoSQL的诞生

现在 - 2010年代，NoSQL开始了最新一轮尝试，试图推翻关系模型的统治地位。“NoSQL”这个名字让人遗憾，因为实际上它并没有涉及到任何特定的技术。最初它只是作为一个醒目的Twitter标签，用在2009年一个关于分布式，非关系数据库上的开源聚会上。无论如何，这个术语触动了某些神经，并迅速在网络创业社区内外传播开来。好些有趣的数据库系统现在都与*#NoSQL#*标签相关联，并且NoSQL被追溯性地重新解释为不仅是**SQL**（**Not Only SQL**）【4】。

采用NoSQL数据库的背后有几个驱动因素，其中包括：

- 需要比关系数据库更好的可扩展性，包括非常大的数据集或非常高的写入吞吐量
- 相比商业数据库产品，免费和开源软件更受偏爱。
- 关系模型不能很好地支持一些特殊的查询操作
- 受挫于关系模型的限制性，渴望一种更具多动态性与表现力的数据模型【5】

不同的应用程序有不同的需求，一个用例的最佳技术选择可能不同于另一个用例的最佳技术选择。因此，在可预见的未来，关系数据库似乎可能会继续与各种非关系数据库一起使用 - 这种想法有时也被称为混合持久化（**polyglot persistence**）

对象关系不匹配

目前大多数应用程序开发都使用面向对象的编程语言来开发，这导致了对SQL数据模型的普遍批评：如果数据存储在关系表中，那么需要一个笨拙的转换层，处于应用程序代码中的对象和表，行，列的数据库模型之间。模型之间的不连贯有时被称为阻抗不匹配（**impedance mismatch**）[i]。

> i. 一个从电子学借用的术语。每个电路的输入和输出都有一定的阻抗（交流电阻）。当你将一个电路的输出连接到另一个电路的输入时，如果两个电路的输出和输入阻抗匹配，则连接上的功率传输将被最大化。阻抗不匹配会导致信号反射及其他问题。↩

像ActiveRecord和Hibernate这样的对象关系映射（**object-relational mapping, ORM**）框架可以减少这个转换层所需的样板代码的数量，但是它们不能完全隐藏这两个模型之间的差异。

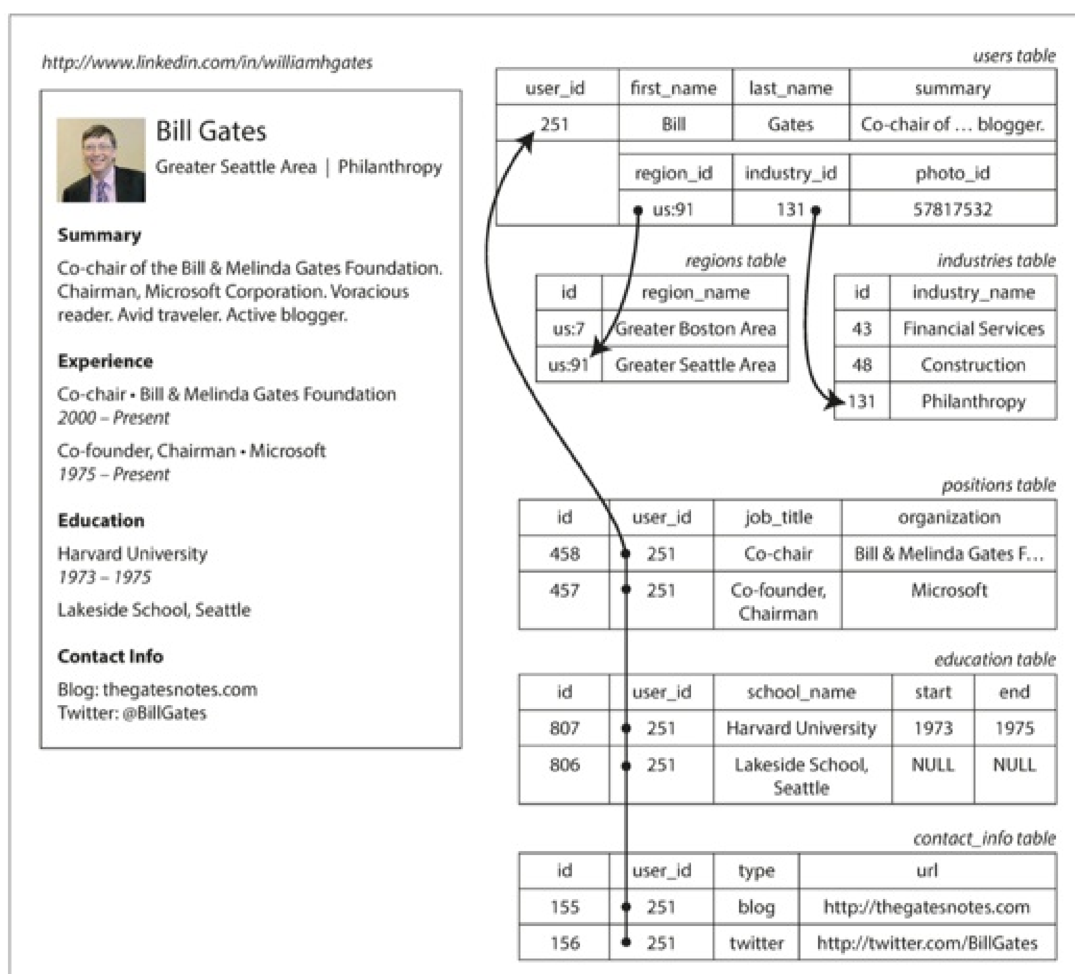


图**2-1** 使用关系型模式来表示领英简介

例如，图2-1展示了如何在关系模式中表示简历（一个LinkedIn简介）。整个简介可以通过一个唯一的标识符 `user_id` 来标识。像 `first_name` 和 `last_name` 这样的字段每个用户只出现一次，所以可以在User表上将其建模为列。但是，大多数人在职业生涯中拥有多于一份的工作，人们可能有不同样的教育阶段和任意数量的联系信息。从用户到这些项目之间存在一对多的关系，可以用多种方式来表示：

- 传统SQL模型（SQL：1999之前）中，最常见的规范化表示形式是将职位，教育和联系信息放在单独的表中，对User表提供外键引用，如图2-1所示。
- 后续的SQL标准增加了对结构化数据类型和XML数据的支持;这允许将多值数据存储在单行内，并支持在这些文档内查询和索引。这些功能在Oracle，IBM DB2，MS SQL Server和PostgreSQL中都有不同程度的支持【6,7】。JSON数据类型也得到多个数据库的支持，包括IBM DB2，MySQL和PostgreSQL 【8】。
- 第三种选择是将职业，教育和联系信息编码为JSON或XML文档，将其存储在数据库的文本列中，并让应用程序解析其结构和内容。这种配置下，通常不能使用数据库来查询该编码列中的值。

对于一个像简历这样自包含文档的数据结构而言，JSON表示是非常合适的：参见例2-1。JSON比XML更简单。面向文档的数据库（如MongoDB 【9】，RethinkDB 【10】，CouchDB 【11】和Espresso【12】）支持这种数据模型。 例**2-1.** 用**JSON**文档表示一个

LinkedIn简介

```json
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {
      "job_title": "Co-chair",
      "organization": "Bill & Melinda Gates Foundation"
    },
    {
      "job_title": "Co-founder, Chairman",
      "organization": "Microsoft"
    }
  ],
  "education": [
    {
      "school_name": "Harvard University",
      "start": 1973,
      "end": 1975
    },
    {
      "school_name": "Lakeside School, Seattle",
      "start": null,
      "end": null
    }
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

有一些开发人员认为JSON模型减少了应用程序代码和存储层之间的阻抗不匹配。不过，正如我们将在第4章中看到的那样，JSON作为数据编码格式也存在问题。缺乏一个模式往往被认为是一个优势;我们将在“文档模型中的模式灵活性”中讨论这个问题。

JSON表示比图2-1中的多表模式具有更好的局部性（**locality**）。如果在前面的关系型示例中获取简介，那需要执行多个查询（通过 `user_id` 查询每个表），或者在User表与其下属表之间混乱地执行多路连接。而在JSON表示中，所有相关信息都在同一个地方，一个查询就足够了。

从用户简介文件到用户职位，教育历史和联系信息，这种一对多关系隐含了数据中的一个树状结构，而JSON表示使得这个树状结构变得明确（见图2-2）。

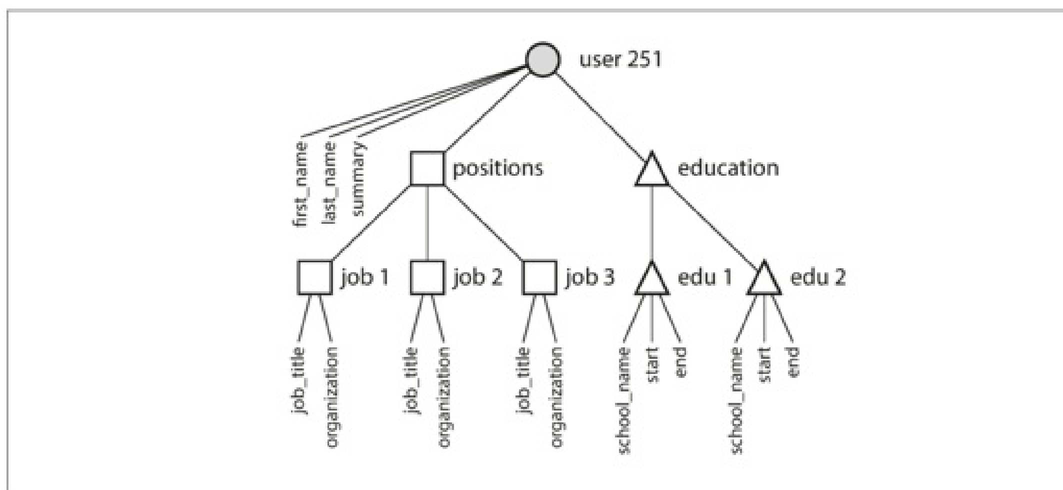


图**2-2** 一对多关系构建了一个树结构

多对一和多对多的关系

在上一节的例2-1中，`region_id` 和 `industry_id` 是以ID，而不是纯字符串“Greater Seattle Area”和“Philanthropy”的形式给出的。为什么？

如果用户界面用一个自由文本字段来输入区域和行业，那么将他们存储为纯文本字符串是合理的。另一方式是给出地理区域和行业的标准化的列表，并让用户从下拉列表或自动填充器中进行选择，其优势如下：

- 各个简介之间样式和拼写统一
- 避免歧义（例如，如果有几个同名的城市）
- 易于更新——名称只存储在一个地方，如果需要更改（例如，由于政治事件而改变城市名称），很容易进行全面更新。
- 本地化支持——当网站翻译成其他语言时，标准化的列表可以被本地化，使得地区和行业可以使用用户的语言来显示
- 更好的搜索——例如，搜索华盛顿州的慈善家就会匹配这份简介，因为地区列表可以编码记录西雅图在华盛顿这一事实（从“Greater Seattle Area”这个字符串中看不出来）

存储ID还是文本字符串，这是个副本（**duplication**）问题。当使用ID时，对人类有意义的信息（比如单词：Philanthropy）只存储在一处，所有引用它的地方使用ID（ID只在数据库中有意义）。当直接存储文本时，对人类有意义的信息会复制在每处使用记录中。

使用ID的好处是，ID对人类没有任何意义，因而永远不需要改变：ID可以保持不变，即使它标识的信息发生变化。任何对人类有意义的东西都可能需要在将来某个时候改变——如果这些信息被复制，所有的冗余副本都需要更新。这会导致写入开销，也存在不一致的风险（一些副本被更新了，还有些副本没有被更新）。去除此类重复是数据库规范化（**normalization**）的关键思想。[ii]

ii

> ii. 关于关系模型的文献区分了几种不同的规范形式，但这些区别几乎没有实际意义。一个经验法则是，如果重复存储了可以存储在一个地方的值，则模式就不是规范化（normalized）的。↩

> 数据库管理员和开发人员喜欢争论规范化和非规范化，让我们暂时保留判断吧。在本书的第三部分，我们将回到这个话题，探讨系统的方法用以处理缓存，非规范化和派生数据。

不幸的是，对这些数据进行规范化需要多对一的关系（许多人生活在一个特定的地区，许多人在一个特定的行业工作），这与文档模型不太吻合。在关系数据库中，通过ID来引用其他表中的行是正常的，因为连接很容易。在文档数据库中，一对多树结构没有必要用连接，对连接的支持通常很弱[iii]。

> iii. 在撰写本文时，RethinkDB支持连接，MongoDB不支持连接，而CouchDB只支持预先声明的视图。↩

如果数据库本身不支持连接，则必须在应用程序代码中通过对数据库进行多个查询来模拟连接。（在这种情况中，地区和行业的列表可能很小，改动很少，应用程序可以简单地将其保存在内存中。不过，执行连接的工作从数据库被转移到应用程序代码上。

此外，即便应用程序的最初版本适合无连接的文档模型，随着功能添加到应用程序中，数据会变得更加互联。例如，考虑一下对简历例子进行的一些修改：

组织和学校作为实体

在前面的描述中，`organization`（用户工作的公司）和`school_name`（他们学习的地方）只是字符串。也许他们应该是对实体的引用呢？然后，每个组织，学校或大学都可以拥有自己的网页（标识，新闻提要等）。每个简历可以链接到它所提到的组织和学校，并且包括他们的图标和其他信息（参见图2-3，来自LinkedIn的一个例子）。

推荐

假设你想添加一个新的功能：一个用户可以为另一个用户写一个推荐。在用户的简历上显示推荐，并附上推荐用户的姓名和照片。如果推荐人更新他们的照片，那他们写的任何建议都需要显示新的照片。因此，推荐应该拥有作者个人简介的引用。

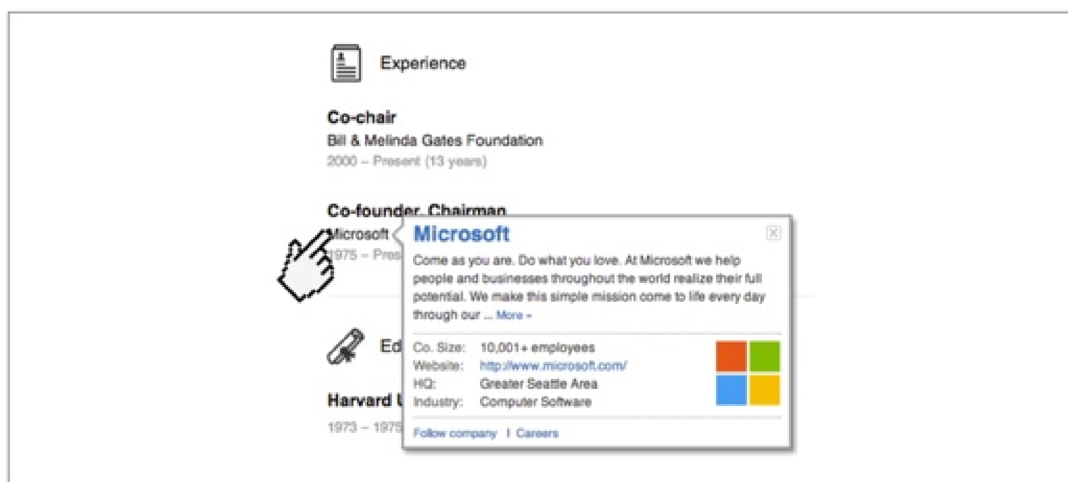


图**2-3** 公司名不仅是字符串，还是一个指向公司实体的链接（**LinkedIn**截图）

图2-4阐明了这些新功能需要如何使用多对多关系。每个虚线矩形内的数据可以分组成一个文档，但是对单位，学校和其他用户的引用需要表示成引用，并且在查询时需要连接。

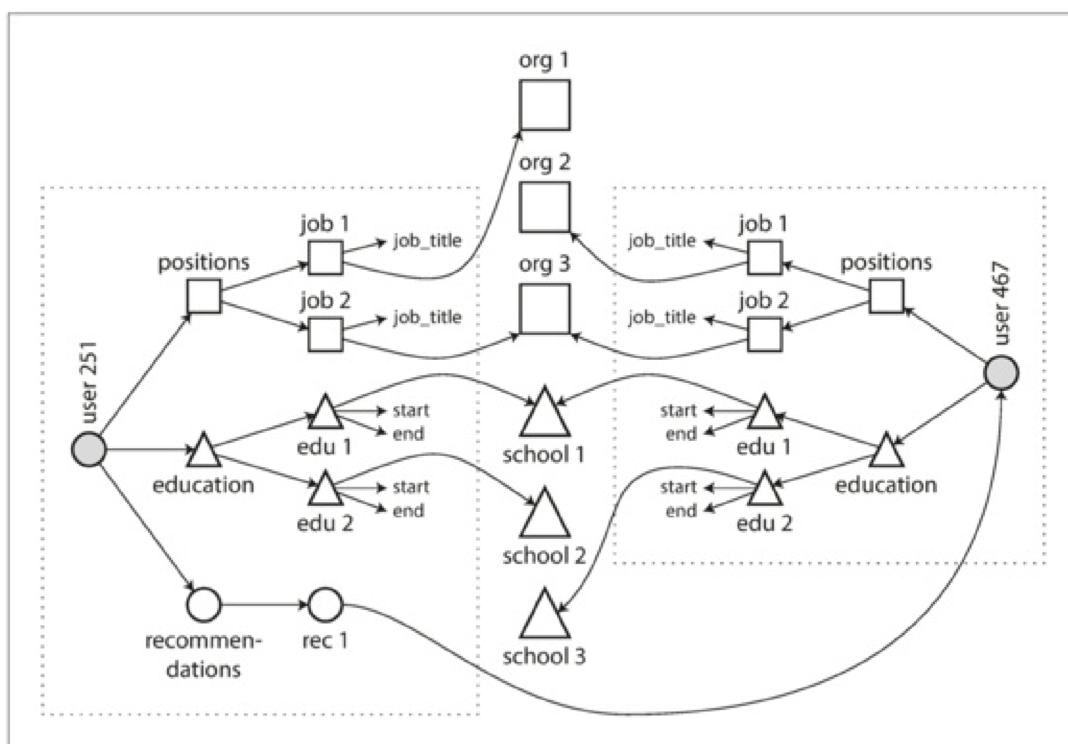


图**2-4** 使用多对多关系扩展简历

文档数据库是否在重蹈覆辙？

在多对多的关系和连接已常规用在关系数据库时，文档数据库和NoSQL重启了辩论：如何最好地在数据库中表示多对多关系。那场辩论可比NoSQL古老得多，事实上，最早可以追溯到计算机化数据库系统。

20世纪70年代最受欢迎的业务数据处理数据库是IBM的信息管理系统（IMS），最初是为了阿波罗太空计划的库存管理而开发的，并于1968年有了首次商业发布【13】。目前它仍在使用和维护，运行在IBM大型机的OS/390上【14】。

IMS的设计中使用了一个相当简单的数据模型，称为层次模型（**hierarchical model**），它与文档数据库使用的JSON模型有一些惊人的相似之处【2】。它将所有数据表示为嵌套在记录中的记录树，这很像图2-2的JSON结构。

同文档数据库一样，IMS能良好处理一对多的关系，但是很难应对多对多的关系，并且不支持连接。开发人员必须决定是否复制（非规范化）数据或手动解决从一个记录到另一个记录的引用。这些二十世纪六七十年代的问题与现在开发人员遇到的文档数据库问题非常相似【15】。

那时人们提出了各种不同的解决方案来解决层次模型的局限性。其中最突出的两个是关系模型（**relational model**）（它变成了SQL，统治了世界）和网络模型（**network model**）（最初很受关注，但最终变得冷门）。这两个阵营之间的“大辩论”在70年代持续了很久时间【2】。

那两个模式解决的问题与当前的问题相关，因此值得简要回顾一下那场辩论。

网络模型

网络模型由一个称为数据系统语言会议（CODASYL）的委员会进行了标准化，并被数个不同的数据库商实现;它也被称为CODASYL模型【16】。

CODASYL模型是层次模型的推广。在层次模型的树结构中，每条记录只有一个父节点；在网络模式中，每条记录可能有多个父节点。例如，“Greater Seattle Area”地区可能是一条记录，每个居住在该地区的用户都可以与之相关联。这允许对多对一和多对多的关系进行建模。

网络模型中记录之间的链接不是外键，而更像编程语言中的指针（同时仍然存储在磁盘上）。访问记录的唯一方法是跟随从根记录起沿这些链路所形成的路径。这被称为访问路径（**access path**）。

最简单的情况下，访问路径类似遍历链表：从列表头开始，每次查看一条记录，直到找到所需的记录。但在多对多关系的情况中，数条不同的路径可以到达相同的记录，网络模型的程序员必须跟踪这些不同的访问路径。

CODASYL中的查询是通过利用遍历记录列和跟随访问路径表在数据库中移动游标来执行的。如果记录有多个父结点（即多个来自其他记录的传入指针），则应用程序代码必须跟踪所有的各种关系。甚至CODASYL委员会成员也承认，这就像在n维数据空间中进行导航【17】。

尽管手动选择访问路径够能最有效地利用20世纪70年代非常有限的硬件功能（如磁带驱动器，其搜索速度非常慢），但这使得查询和更新数据库的代码变得复杂不灵活。无论是分层还是网络模型，如果你没有所需数据的路径，就会陷入困境。你可以改变访问路径，但是必

须浏览大量手写数据库查询代码，并重写来处理新的访问路径。更改应用程序的数据模型是很难的。

关系模型

相比之下，关系模型做的就是将所有的数据放在光天化日之下：一个关系（表）只是一个元组（行）的集合，仅此而已。如果你想读取数据，它没有迷宫似的嵌套结构，也没有复杂的访问路径。你可以选中符合任意条件的行，读取表中的任何或所有行。你可以通过指定某些列作为匹配关键字来读取特定行。你可以在任何表中插入一个新的行，而不必担心与其他表的外键关系[iv]。

> iv. 外键约束允许对修改约束，但对于关系模型这并不是必选项。即使有约束，外键连接在查询时执行，而在CODASYL中，连接在插入时高效完成。↩

在关系数据库中，查询优化器自动决定查询的哪些部分以哪个顺序执行，以及使用哪些索引。这些选择实际上是“访问路径”，但最大的区别在于它们是由查询优化器自动生成的，而不是由程序员生成，所以我们很少需要考虑它们。

如果想按新的方式查询数据，你可以声明一个新的索引，查询会自动使用最合适的那些索引。无需更改查询来利用新的索引。（请参阅“用于数据的查询语言”。）关系模型因此使添加应用程序新功能变得更加容易。

关系数据库的查询优化器是复杂的，已耗费了多年的研究和开发精力【18】。关系模型的一个关键洞察是：只需构建一次查询优化器，随后使用该数据库的所有应用程序都可以从中受益。如果你没有查询优化器的话，那么为特定查询手动编写访问路径比编写通用优化器更容易——不过从长期看通用解决方案更好。

与文档数据库相比

在一个方面，文档数据库还原为层次模型：在其父记录中存储嵌套记录（图2-1中的一对多关系，如 `positions` ，`education` 和 `contact_info` ），而不是在单独的表中。

但是，在表示多对一和多对多的关系时，关系数据库和文档数据库并没有根本的不同：在这两种情况下，相关项目都被一个唯一的标识符引用，这个标识符在关系模型中被称为外键，在文档模型中称为文档引用【9】。该标识符在读取时通过连接或后续查询来解析。迄今为止，文档数据库没有走CODASYL的老路。

关系型数据库与文档数据库在今日的对比

将关系数据库与文档数据库进行比较时，可以考虑许多方面的差异，包括它们的容错属性（参阅第5章）和处理并发性（参阅第7章）。本章将只关注数据模型中的差异。

支持文档数据模型的主要论据是架构灵活性，因局部性而拥有更好的性能，以及对于某些应用程序而言更接近于应用程序使用的数据结构。关系模型通过为连接提供更好的支持以及支持多对一和多对多的关系来反击。

哪个数据模型更方便写代码？

如果应用程序中的数据具有类似文档的结构（即，一对多关系树，通常一次性加载整个树），那么使用文档模型可能是一个好主意。将类似文档的结构分解成多个表（如图2-1中的 `positions` ， `education` 和 `contact_info` ）的关系技术可能导致繁琐的模式和不必要的复杂的应用程序代码。

文档模型有一定的局限性：例如，不能直接引用文档中的嵌套的项目，而是需要说“用户251的位置列表中的第二项”（很像分层模型中的访问路径）。但是，只要文件嵌套不太深，这通常不是问题。

文档数据库对连接的糟糕支持也许或也许不是一个问题，这取决于应用程序。例如，分析应用程可能永远不需要多对多的关系，如果它使用文档数据库来记录何事发生于何时【19】。

但是，如果你的应用程序确实使用多对多关系，那么文档模型就没有那么吸引人了。通过反规范化可以减少对连接的需求，但是应用程序代码需要做额外的工作来保持数据的一致性。通过向数据库发出多个请求，可以在应用程序代码中模拟连接，但是这也将复杂性转移到应用程序中，并且通常比由数据库内的专用代码执行的连接慢。在这种情况下，使用文档模型会导致更复杂的应用程序代码和更差的性能【15】。

很难说在一般情况下哪个数据模型让应用程序代码更简单；它取决于数据项之间存在的关系种类。对于高度相联的数据，选用文档模型是糟糕的，选用关系模型是可接受的，而选用图形模型（参见“图数据模型”）是最自然的。

文档模型中的架构灵活性

大多数文档数据库以及关系数据库中的JSON支持都不会强制文档中的数据采用何种模式。关系数据库的XML支持通常带有可选的模式验证。没有模式意味着可以将任意的键和值添加到文档中，并且当读取时，客户端对无法保证文档可能包含的字段。

文档数据库有时称为无模式（**schemaless**），但这具有误导性，因为读取数据的代码通常假定某种结构——即存在隐式模式，但不由数据库强制执行【20】。一个更精确的术语是读时模式（**schema-on-read**）（数据的结构是隐含的，只有在数据被读取时才被解释），相应的是写时模式（**schema-on-write**）（传统的关系数据库方法中，模式明确，且数据库确保所有的数据都符合其模式）【21】。

读时模式类似于编程语言中的动态（运行时）类型检查，而写时模式类似于静态（编译时）类型检查。就像静态和动态类型检查的相对优点具有很大的争议性一样【22】，数据库中模式的强制性是一个具有争议的话题，一般来说没有正确或错误的答案。

在应用程序想要改变其数据格式的情况下，这些方法之间的区别尤其明显。例如，假设你把每个用户的全名存储在一个字段中，而现在想分别存储名字和姓氏【23】。在文档数据库中，只需开始写入具有新字段的新文档，并在应用程序中使用代码来处理读取旧文档的情况。例如：

```
if (user && user.name && !user.first_name) {
    // Documents written before Dec 8, 2013 don't have first_name
    user.first_name = user.name.split(" ")[0];
}
```

另一方面，在“静态类型”数据库模式中，通常会执行以下迁移（**migration**）操作：

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name = split_part(name, ' ', 1);        -- PostgreSQL
UPDATE users SET first_name = substring_index(name, ' ', 1);     -- MySQL
```

模式变更的速度很慢，而且要求停运。它的这种坏名誉并不是完全应得的：大多数关系数据库系统可在几毫秒内执行 `ALTER TABLE` 语句。MySQL是一个值得注意的例外，它执行 `ALTER TABLE` 时会复制整个表，这可能意味着在更改一个大型表时会花费几分钟甚至几个小时的停机时间，尽管存在各种工具来解决这个限制【24,25,26】。

大型表上运行 `UPDATE` 语句在任何数据库上都可能会很慢，因为每一行都需要重写。要是不可接受的话，应用程序可以将 `first_name` 设置为默认值 `NULL` ，并在读取时再填充，就像使用文档数据库一样。

读时模式更具优势，当由于某种原因（例如，数据是异构的）集合中的项目并不都具有相同的结构时。例如，因为：

- 存在许多不同类型的对象，将每种类型的对象放在自己的表中是不现实的。
- 数据的结构由外部系统决定。你无法控制外部系统且它随时可能变化。

在这样的情况下，模式的坏处远大于它的帮助，无模式文档可能是一个更加自然的数据模型。但是，要是所有记录都具有相同的结构，那么模式是记录并强制这种结构的有效机制。第四章将更详细地讨论模式和模式演化。

查询的数据局部性

文档通常以单个连续字符串形式进行存储，编码为JSON，XML或其二进制变体（如MongoDB的BSON）。如果应用程序经常需要访问整个文档（例如，将其渲染至网页），那么存储局部性会带来性能优势。如果将数据分割到多个表中（如图2-1所示），则需要进行多次索引查找才能将其全部检索出来，这可能需要更多的磁盘查找并花费更多的时间。

局部性仅仅适用于同时需要文档绝大部分内容的情况。数据库通常需要加载整个文档，即使只访问其中的一小部分，这对于大型文档来说是很浪费的。更新文档时，通常需要整个重写。只有不改变文档大小的修改才可以容易地原地执行。因此，通常建议保持相对小的文档，并避免增加文档大小的写入【9】。这些性能限制大大减少了文档数据库的实用场景。

值得指出的是，为了局部性而分组集合相关数据的想法并不局限于文档模型。例如，Google的Spanner数据库在关系数据模型中提供了同样的局部性属性，允许模式声明一个表的行应该交错（嵌套）在父表内【27】。Oracle类似地允许使用一个称为多表索引集群表（**multi-**

table index cluster tables）的类似特性【28】。Bigtable数据模型（用于Cassandra和HBase）中的列族（**column-family**）概念与管理局部性的目的类似【29】。

在第3章将还会看到更多关于局部性的内容。

文档和关系数据库的融合

自2000年代中期以来，大多数关系数据库系统（MySQL除外）都已支持XML。这包括对XML文档进行本地修改的功能，以及在XML文档中进行索引和查询的功能。这允许应用程序使用那种与文档数据库应当使用的非常类似的数据模型。

从9.3版本开始的PostgreSQL 【8】，从5.7版本开始的MySQL以及从版本10.5开始的IBM DB2 [30]也对JSON文档提供了类似的支持级别。鉴于用在Web APIs的JSON流行趋势，其他关系数据库很可能会跟随他们的脚步并添加JSON支持。

在文档数据库中，RethinkDB在其查询语言中支持类似关系的连接，一些MongoDB驱动程序可以自动解析数据库引用（有效地执行客户端连接，尽管这可能比在数据库中执行的连接慢，需要额外的网络往返，并且优化更少）。

随着时间的推移，关系数据库和文档数据库似乎变得越来越相似，这是一件好事：数据模型相互补充[V]，如果一个数据库能够处理类似文档的数据，并能够对其执行关系查询，那么应用程序就可以使用最符合其需求的功能组合。

关系模型和文档模型的混合是未来数据库一条很好的路线。

> [V]. Codd对关系模型【1】的原始描述实际上允许在关系模式中与JSON文档非常相似。他称之为非简单域（**nonsimple domains**）。这个想法是，一行中的值不一定是一个像数字或字符串一样的原始数据类型，也可以是一个嵌套的关系（表），因此可以把一个任意嵌套的树结构作为一个值，这很像30年后添加到SQL中的JSON或XML支持。↩

数据查询语言

当引入关系模型时，关系模型包含了一种查询数据的新方法：SQL是一种声明式查询语言，而IMS和CODASYL使用命令式代码来查询数据库。那是什么意思？

许多常用的编程语言是命令式的。例如，给定一个动物物种的列表，返回列表中的鲨鱼可以这样写：

```
function getSharks() {
    var sharks = [];
    for (var i = 0; i < animals.length; i++) {
        if (animals[i].family === "Sharks") {
            sharks.push(animals[i]);
        }
    }
    return sharks;
}
```

在关系代数中：

$$ sharks = σ_{family = "sharks"}(animals)

$$ σ（希腊字母西格玛）是选择操作符，只返回符合条件的动物， `family="shark"` 。

定义SQL时，它紧密地遵循关系代数的结构：

```
SELECT * FROM animals WHERE family ='Sharks';
```

命令式语言告诉计算机以特定顺序执行某些操作。可以想象一下，逐行地遍历代码，评估条件，更新变量，并决定是否再循环一遍。

在声明式查询语言（如SQL或关系代数）中，你只需指定所需数据的模式 - 结果必须符合哪些条件，以及如何将数据转换（例如，排序，分组和集合） - 但不是如何实现这一目标。数据库系统的查询优化器决定使用哪些索引和哪些连接方法，以及以何种顺序执行查询的各个部分。

声明式查询语言是迷人的，因为它通常比命令式API更加简洁和容易。但更重要的是，它还隐藏了数据库引擎的实现细节，这使得数据库系统可以在无需对查询做任何更改的情况下进行性能提升。

例如，在本节开头所示的命令代码中，动物列表以特定顺序出现。如果数据库想要在后台回收未使用的磁盘空间，则可能需要移动记录，这会改变动物出现的顺序。数据库能否安全地执行，而不会中断查询？

SQL示例不确保任何特定的顺序，因此不在意顺序是否改变。但是如果查询用命令式的代码来写的话，那么数据库就永远不可能确定代码是否依赖于排序。SQL相当有限的功能性为数据库提供了更多自动优化的空间。

最后，声明式语言往往适合并行执行。现在，CPU的速度通过内核的增加变得更快，而不是以比以前更高的时钟速度运行【31】。命令代码很难在多个内核和多个机器之间并行化，因为它指定了指令必须以特定顺序执行。声明式语言更具有并行执行的潜力，因为它们仅指定结果的模式，而不指定用于确定结果的算法。在适当情况下，数据库可以自由使用查询语言的并行实现【32】。

Web上的声明式查询

声明式查询语言的优势不仅限于数据库。为了说明这一点，让我们在一个完全不同的环境中比较声明式和命令式方法：一个Web浏览器。

假设你有一个关于海洋动物的网站。用户当前正在查看鲨鱼页面，因此你将当前所选的导航项目“鲨鱼”标记为当前选中项目。

```
<ul>
    <li class="selected">
        <p>Sharks</p>
        <ul>
            <li>Great White Shark</li>
            <li>Tiger Shark</li>
            <li>Hammerhead Shark</li>
        </ul>
    </li>
    <li><p>Whales</p>
        <ul>
            <li>Blue Whale</li>
            <li>Humpback Whale</li>
            <li>Fin Whale</li>
        </ul>
    </li>
</ul>
```

现在想让当前所选页面的标题具有一个蓝色的背景，以便在视觉上突出显示。使用CSS实现起来非常简单：

```
li.selected > p {
    background-color: blue;
}
```

这里的CSS选择器 li.selected> p 声明了我们想要应用蓝色样式的元素的模式：即其直接父元素是具有 selected CSS类的 <li> 元素的所有 <p> 元素。示例中的元素 <p> Sharks </p> 匹配此模式，但 <p> Whales </p> 不匹配，因为其 <li> 父元素缺少 class =“selected” 。

如果使用XSL而不是CSS，你可以做类似的事情：

```
<xsl:template match="li[@class='selected']/p">
    <fo:block background-color="blue">
        <xsl:apply-templates/>
    </fo:block>
</xsl:template>
```

这里的XPath表达式 li[@class='selected']/p 相当于上例中的CSS选择器 li.selected> p 。CSS和XSL的共同之处在于，它们都是用于指定文档样式的声明式语言。

想象一下，必须使用命令式方法的情况会是如何。在Javascript中，使用文档对象模型（**DOM**）API，其结果可能如下所示：

```
var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
    if (liElements[i].className === "selected") {
        var children = liElements[i].childNodes;
        for (var j = 0; j < children.length; j++) {
            var child = children[j];
            if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
                child.setAttribute("style", "background-color: blue");
            }
        }
    }
}
```

这段JavaScript代码命令式地将元素设置为蓝色背景，但是代码看起来很糟糕。不仅比CSS和XSL等价物更长，更难理解，而且还有一些严重的问题：

- 如果选定的类被移除（例如，因为用户点击了不同的页面），即使代码重新运行，蓝色背景也不会被移除 - 因此该项目将保持突出显示，直到整个页面被重新加载。使用CSS，浏览器会自动检测 `li.selected> p` 规则何时不再适用，并在选定的类被移除后立即移除蓝色背景。
- 如果你想要利用新的API（例如 `document.getElementsBy ClassName（“selected”` ）甚至 `document.evaluate()` ）来提高性能，则必须重写代码。另一方面，浏览器供应商可以在不破坏兼容性的情况下提高CSS和XPath的性能。

在Web浏览器中，使用声明式CSS样式比使用JavaScript命令式地操作样式要好得多。类似地，在数据库中，使用像SQL这样的声明式查询语言比使用命令式查询API要好得多[vi]。

> [vi]. vi IMS和CODASYL都使用命令式API。应用程序通常使用COBOL代码遍历数据库中的记录，一次一条记录【2,16】。↩

MapReduce查询

MapReduce是一个由Google推广的编程模型，用于在多台机器上批量处理大规模的数据【33】。一些NoSQL数据存储（包括MongoDB和CouchDB）支持有限形式的MapReduce，作为在多个文档中执行只读查询的机制。

MapReduce将第10章中有更详细的描述。现在我们将简要讨论一下MongoDB使用的模型。

MapReduce既不是一个声明式的查询语言，也不是一个完全命令式的查询API，而是处于两者之间：查询的逻辑用代码片断来表示，这些代码片段会被处理框架重复性调用。它基于 `map` （也称为 `collect` ）和 `reduce` （也称为 `fold` 或 `inject` ）函数，两个函数存在于许多函数式编程语言中。

最好举例来解释MapReduce模型。假设你是一名海洋生物学家，每当你看到海洋中的动物时，你都会在数据库中添加一条观察记录。现在你想生成一个报告，说明你每月看到多少鲨鱼。

在PostgreSQL中，你可以像这样表述这个查询：

```
SELECT
    date_trunc('month', observation_timestamp) AS observation_month,
    sum(num_animals)                           AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

`date_trunc('month'，timestamp)` 函数用于确定包含 `timestamp` 的日历月份，并返回代表该月份开始的另一个时间戳。换句话说，它将时间戳舍入成最近的月份。

这个查询首先过滤观察记录，以只显示鲨鱼家族的物种，然后根据它们发生的日历月份对观察记录果进行分组，最后将在该月的所有观察记录中看到的动物数目加起来。

同样的查询用MongoDB的MapReduce功能可以按如下来表述：

```
db.observations.mapReduce(function map() {
        var year = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals);
    },
    function reduce(key, values) {
        return Array.sum(values);
    },
    {
        query: {
          family: "Sharks"
        },
        out: "monthlySharkReport"
    });
```

- 可以声明式地指定只考虑鲨鱼种类的过滤器（这是一个针对MapReduce的特定于MongoDB的扩展）。
- 每个匹配查询的文档都会调用一次JavaScript函数 `map` ，将 `this` 设置为文档对象。
- `map` 函数发出一个键（包括年份和月份的字符串，如 `"2013-12"` 或 `"2014-1"` ）和一个值（该观察记录中的动物数量）。
- `map` 发出的键值对按键来分组。对于具有相同键（即，相同的月份和年份）的所有键值对，调用一次 `reduce` 函数。
- `reduce` 函数将特定月份内所有观测记录中的动物数量相加。
- 将最终的输出写入到 `monthlySharkReport` 集合中。

例如，假设 `observations` 集合包含这两个文档：

```
{
  observationTimestamp: Date.parse(  "Mon, 25 Dec 1995 12:34:56 GMT"),
  family: "Sharks",
  species: "Carcharodon carcharias",
  numAnimals: 3
{
}
  observationTimestamp: Date.parse("Tue, 12 Dec 1995 16:17:18 GMT"),
  family: "Sharks",
  species:    "Carcharias taurus",
  numAnimals: 4
}
```

对每个文档都会调用一次 `map` 函数，结果将是 `emit("1995-12",3)` 和 `emit("1995-12",4)` 。随后，以 `reduce("1995-12",[3,4])` 调用 `reduce` 函数，将返回 `7` 。

map和reduce函数在功能上有所限制：它们必须是纯函数，这意味着它们只使用传递给它们的数据作为输入，它们不能执行额外的数据库查询，也不能有任何副作用。这些限制允许数据库以任何顺序运行任何功能，并在失败时重新运行它们。然而，map和reduce函数仍然是强大的：它们可以解析字符串，调用库函数，执行计算等等。

MapReduce是一个相当底层的编程模型，用于计算机集群上的分布式执行。像SQL这样的更高级的查询语言可以用一系列的MapReduce操作来实现（见第10章），但是也有很多不使用MapReduce的分布式SQL实现。请注意，SQL中没有任何内容限制它在单个机器上运行，而MapReduce在分布式查询执行上没有垄断权。

能够在查询中使用JavaScript代码是高级查询的一个重要特性，但这不限于MapReduce，一些SQL数据库也可以用JavaScript函数进行扩展【34】。

MapReduce的一个可用性问题是，必须编写两个密切合作的JavaScript函数，这通常比编写单个查询更困难。此外，声明式查询语言为查询优化器提供了更多机会来提高查询的性能。基于这些原因，MongoDB 2.2添加了一种叫做聚合管道的声明式查询语言的支持【9】。用这种语言表述鲨鱼计数查询如下所示：

```
db.observations.aggregate([
  { $match: { family: "Sharks" } },
  { $group: {
    _id: {
      year:  { $year:  "$observationTimestamp" },
      month: { $month: "$observationTimestamp" }
    },
    totalAnimals: { $sum: "$numAnimals" } }}
]);
```

聚合管道语言与SQL的子集具有类似表现力，但是它使用基于JSON的语法而不是SQL的英语句子式语法; 这种差异也许是口味问题。这个故事的寓意是NoSQL系统可能会发现自己意外地重新发明了SQL，尽管带着伪装。

图数据模型

如我们之前所见，多对多关系是不同数据模型之间具有区别性的重要特征。如果你的应用程序大多数的关系是一对多关系（树状结构化数据），或者大多数记录之间不存在关系，那么使用文档模型是合适的。

但是，要是多对多关系在你的数据中很常见呢？关系模型可以处理多对多关系的简单情况，但是随着数据之间的连接变得更加复杂，将数据建模为图形显得更加自然。

一个图由两种对象组成：顶点（**vertices**）（也称为节点（**nodes**）或实体（**entities**）），和边（**edges**）（也称为关系（**relationships**）或弧（**arcs**））。多种数据可以被建模为一个图形。典型的例子包括：

社交图谱

顶点是人，边指示哪些人彼此认识。

网络图谱

顶点是网页，边缘表示指向其他页面的HTML链接。

公路或铁路网络

顶点是交叉路口，边线代表它们之间的道路或铁路线。

可以将那些众所周知的算法运用到这些图上：例如，汽车导航系统搜索道路网络中两点之间的最短路径，PageRank可以用在网络图上来确定网页的流行程度，从而确定该网页在搜索结果中的排名。

在刚刚给出的例子中，图中的所有顶点代表了相同类型的事物（人，网页或交叉路口）。不过，图并不局限于这样的同类数据：同样强大地是，图提供了一种一致的方式，用来在单个数据存储中存储完全不同类型的对象。例如，Facebook维护一个包含许多不同类型的顶点和边的单个图：顶点表示人，地点，事件，签到和用户的评论;边缘表示哪些人是彼此的朋友，哪个签到发生在何处，谁评论了哪条消息，谁参与了哪个事件，等等【35】。

在本节中，我们将使用图2-5所示的示例。它可以从社交网络或系谱数据库中获得：它显示了两个人，来自爱达荷州的Lucy和来自法国Beaune的Alain。他们已婚，住在伦敦。

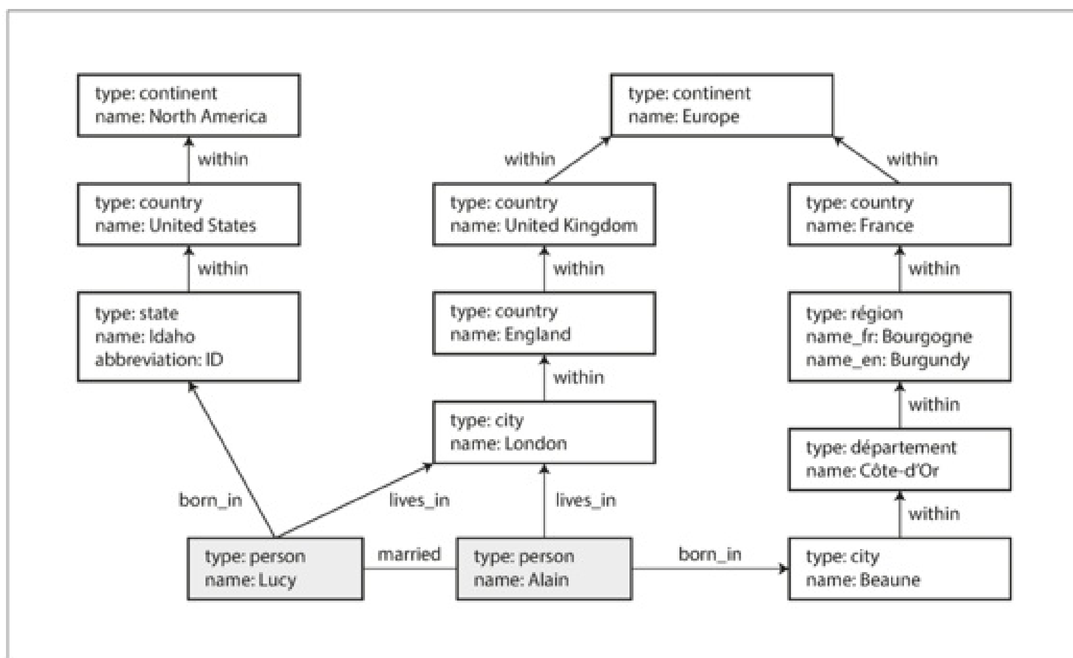


图**2-5** 图数据结构示例（框代表顶点，箭头代表边）

有几种不同但相关的方法用来构建和查询图表中的数据。在本节中，我们将讨论属性图模型（由Neo4j，Titan和InfiniteGraph实现）和三元组存储（triple-store）模型（由Datomic，AllegroGraph等实现）。我们将查看图的三种声明式查询语言：Cypher，SPARQL和Datalog。除此之外，还有像Gremlin 【36】这样的图形查询语言和像Pregel这样的图形处理框架（见第10章）。

属性图

在属性图模型中，每个顶点（**vertex**）包括：

- 唯一的标识符
- 一组出边（**outgoing edges**）
- 一组入边（**ingoing edges**）
- 一组属性（键值对）

每条边（**edge**）包括：

- 唯一标识符
- 边的起点/尾部顶点（**tail vertex**）
- 边的终点/头部顶点（**head vertex**）
- 描述两个顶点之间关系类型的标签
- 一组属性（键值对）

可以将图存储看作由两个关系表组成：一个存储顶点，另一个存储边，如例2-2所示（该模式使用PostgreSQL json数据类型来存储每个顶点或每条边的属性）。头部和尾部顶点用来存储每条边；如果你想要一组顶点的输入或输出边，你可以分别通过 `head_vertex` 或 `tail_vertex` 来查询 `edges` 表。

例**2-2** 使用关系模式来表示属性图

```
CREATE TABLE vertices (
  vertex_id  INTEGER PRIMARY KEY,
  properties JSON
);

CREATE TABLE edges (
  edge_id     INTEGER PRIMARY KEY,
  tail_vertex INTEGER REFERENCES vertices (vertex_id),
  head_vertex INTEGER REFERENCES vertices (vertex_id),
  label       TEXT,
  properties  JSON
);

CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

关于这个模型的一些重要方面是：

1. 任何顶点都可以有一条边连接到任何其他顶点。没有模式限制哪种事物可不可以关联。

2. 给定任何顶点，可以高效地找到它的入边和出边，从而遍历图，即沿着一系列顶点的路径前后移动。（这就是为什么例2-2在 `tail_vertex` 和 `head_vertex` 列上都有索引的原因。）
3. 通过对不同类型的关系使用不同的标签，可以在一个图中存储几种不同的信息，同时仍然保持一个清晰的数据模型。

这些特性为数据建模提供了很大的灵活性，如图2-5所示。图中显示了一些传统关系模式难以表达的事情，例如不同国家的不同地区结构（法国有省和州，美国有不同的州和州），国中国的怪事（先忽略主权国家和国家错综复杂的烂摊子），不同的数据粒度（Lucy现在的住所被指定为一个城市，而她的出生地点只是在一个州的级别）。

你可以想象延伸图还能包括许多关于Lucy和Alain，或其他人的其他更多的事实。例如，你可以用它来表示食物过敏（为每个过敏源增加一个顶点，并增加人与过敏源之间的一条边来指示一种过敏情况），并链接到过敏源，每个过敏源具有一组顶点用来显示哪些食物含有哪些物质。然后，你可以写一个查询，找出每个人吃什么是安全的。图表在可演化性是富有优势的：当向应用程序添加功能时，可以轻松扩展图以适应应用程序数据结构的变化。

Cypher查询语言

Cypher是属性图的声明式查询语言，为Neo4j图形数据库而发明【37】。（它是以电影“黑客帝国”中的一个角色开命名的，而与密码术中的密码无关【38】。）

例2-3显示了将图2-5的左边部分插入图形数据库的Cypher查询。可以类似地添加图的其余部分，为了便于阅读而省略。每个顶点都有一个像 `USA` 或 `Idaho` 这样的符号名称，查询的其他部分可以使用这些名称在顶点之间创建边，使用箭头符号： `（Idaho） - [：WITHIN] ->（USA）` 创建一条标记为 `WITHIN` 的边， `Idaho` 为尾节点， `USA` 为头节点。

例**2-3** 将图**2-5**中的数据子集表示为**Cypher**查询

```
CREATE
    (NAmerica:Location {name:'North America', type:'continent'}),
    (USA:Location      {name:'United States', type:'country'  }),
    (Idaho:Location    {name:'Idaho',         type:'state'    }),
    (Lucy:Person       {name:'Lucy' }),
    (Idaho) -[:WITHIN]->  (USA)  -[:WITHIN]-> (NAmerica),
    (Lucy)  -[:BORN_IN]-> (Idaho)
```

当图2-5的所有顶点和边被添加到数据库后，让我们提些有趣的问题：例如，找到所有从美国移民到欧洲的人的名字。更确切地说，这里我们想要找到符合下面条件的所有顶点，并且返回这些顶点的 `name` 属性：该顶点拥有一条连到美国任一位置的 `BORN_IN` 边，和一条连到欧洲的任一位置的 `LIVING_IN` 边。

例2-4展示了如何在Cypher中表达这个查询。在MATCH子句中使用相同的箭头符号来查找图中的模式：`(person) -[:BORN_IN]-> ()` 可以匹配 `BORN_IN` 边的任意两个顶点。该边的尾节点被绑定了变量 `person`，头节点则未被绑定。

例**2-4** 查找所有从美国移民到欧洲的人的**Cypher**查询：

```
MATCH
    (person) -[:BORN_IN]->  () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
    (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

查询按如下来解读：

> 找到满足以下两个条件的所有顶点（称之为person顶点）：
>
> 1. `person` 顶点拥有一条到某个顶点的 `BORN_IN` 出边。从那个顶点开始，沿着一系列 `WITHIN` 出边最终到达一个类型为 `Location`，`name` 属性为 `United States` 的顶点。
>
> 2. `person` 顶点还拥有一条 `LIVES_IN` 出边。沿着这条边，可以通过一系列 `WITHIN` 出边最终到达一个类型为 `Location`，`name` 属性为 `Europe` 的顶点。
>
> 对于这样的 `Person` 顶点，返回其 `name` 属性。

执行这条查询可能会有几种可行的查询路径。这里给出的描述建议首先扫描数据库中的所有人，检查每个人的出生地和居住地，然后只返回符合条件的那些人。

等价地，也可以从两个 `Location` 顶点开始反向地查找。假如 `name` 属性上有索引，则可以高效地找到代表美国和欧洲的两个顶点。然后，沿着所有 `WITHIN` 入边，可以继续查找出所有在美国和欧洲的位置（州，地区，城市等）。最后，查找出那些可以由 `BORN_IN` 或 `LIVES_IN` 入边到那些位置顶点的人。

通常对于声明式查询语言来说，在编写查询语句时，不需要指定执行细节：查询优化程序会自动选择预测效率最高的策略，因此你可以继续编写应用程序的其他部分。

SQL中的图查询

例2-2建议在关系数据库中表示图数据。但是，如果把图数据放入关系结构中，我们是否也可以使用SQL查询它？

答案是肯定的，但有些困难。在关系数据库中，你通常会事先知道在查询中需要哪些连接。在图查询中，你可能需要在找到待查找的顶点之前，遍历可变数量的边。也就是说，连接的数量事先并不确定。

在我们的例子中，这发生在Cypher查询中的 `() -[:WITHIN*0..]-> ()` 规则中。一个人的 `LIVES_IN` 边可以指向任何类型的位置：街道，城市，地区，地区，国家等。城市可以在一个地区，在一个州内的一个地区，在一个国家内的一个州等等。 `LIVES_IN` 边可以直接指向正在查找的位置，或者一个在位置层次结构中隔了数层的位置。

在Cypher中，用 `WITHIN * 0` 非常简洁地表述了上述事实："沿着 `WITHIN` 边，零次或多次"。它很像正则表达式中的 `*` 运算符。

自SQL:1999，查询可变长度遍历路径的思想可以使用称为递归公用表表达式（ `WITH RECURSIVE` 语法）的东西来表示。例2-5显示了同样的查询 - 查找从美国移民到欧洲的人的姓名 - 在SQL使用这种技术（PostgreSQL，IBM DB2，Oracle和SQL Server均支持）来表述。但是，与Cypher相比，其语法非常笨拙。

例**2-5** 与示例**2-4**同样的查询，在**SQL**中使用递归公用表表达式表示

```
WITH RECURSIVE
  -- in_usa 包含所有的美国境内的位置ID
    in_usa(vertex_id) AS (
    SELECT vertex_id FROM vertices WHERE properties ->> 'name' = 'United States'
    UNION
    SELECT edges.tail_vertex FROM edges
      JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
      WHERE edges.label = 'within'
  ),
  -- in_europe 包含所有的欧洲境内的位置ID
    in_europe(vertex_id) AS (
    SELECT vertex_id FROM vertices WHERE properties ->> 'name' = 'Europe'
    UNION
    SELECT edges.tail_vertex FROM edges
      JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
      WHERE edges.label = 'within' ),

  -- born_in_usa 包含了所有类型为Person，且出生在美国的顶点
    born_in_usa(vertex_id) AS (
      SELECT edges.tail_vertex FROM edges
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'born_in' ),

  -- lives_in_europe 包含了所有类型为Person，且居住在欧洲的顶点。
    lives_in_europe(vertex_id) AS (
      SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.label = 'lives_in')

  SELECT vertices.properties ->> 'name'
  FROM vertices
    JOIN born_in_usa ON vertices.vertex_id = born_in_usa.vertex_id
    JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;
```

- 首先，查找 `name` 属性为 `United States` 的顶点，将其作为 `in_use` 顶点的集合的第一个元素。
- 从 `in_use` 集合的顶点出发，沿着所有的 `with_in` 入边，将其尾顶点加入同一集合，不断递归直到所有 `with_in` 入边都被访问完毕。
- 同理，从 `name` 属性为 `Europe` 的顶点出发，建立 `in_europe` 顶点的集合。
- 对于 `in_usa` 集合中的每个顶点，根据 `born_in` 入边来查找出生在美国某个地方的人。
- 同样，对于 `in_europe` 集合中的每个顶点，根据 `lives_in` 入边来查找居住在欧洲的人。

- 最后，把在美国出生的人的集合与在欧洲居住的人的集合相交。

同一个查询，用某一个查询语言可以写成4行，而用另一个查询语言需要29行，这恰恰说明了不同的数据模型是为不同的应用场景而设计的。选择适合应用程序的数据模型非常重要。

三元组存储和SPARQL

三元组存储模式大体上与属性图模型相同，用不同的词来描述相同的想法。不过仍然值得讨论，因为三元组存储有很多现成的工具和语言，这些工具和语言对于构建应用程序的工具箱可能是宝贵的补充。

在三元组存储中，所有信息都以非常简单的三部分表示形式存储（主语，谓语，宾语）。例如，三元组(吉姆, 喜欢 ,香蕉)中，吉姆是主语，喜欢是谓语（动词），香蕉是对象。

三元组的主语相当于图中的一个顶点。而宾语是下面两者之一：

1. 原始数据类型中的值，例如字符串或数字。在这种情况下，三元组的谓语和宾语相当于主语顶点上的属性的键和值。例如，`(lucy, age, 33)` 就像属性 `{“age”：33}` 的顶点lucy。
2. 图中的另一个顶点。在这种情况下，谓语是图中的一条边，主语是其尾部顶点，而宾语是其头部顶点。例如，在 `(lucy, marriedTo, alain)` 中主语和宾语 `lucy` 和 `alain` 都是顶点，并且谓语 `marriedTo` 是连接他们的边的标签。

例2-6显示了与例2-3相同的数据，以称为Turtle的格式（Notation3（N3）【39】）的一个子集形式写成三元组。

例**2-6** 图**2-5**中的数据子集，表示为**Turtle**三元组

```
@prefix : <urn:example:>.
_:lucy     a       :Person.
_:lucy     :name   "Lucy".
_:lucy     :bornIn _:idaho.
_:idaho    a       :Location.
_:idaho    :name   "Idaho".
_:idaho    :type   "state".
_:idaho    :within _:usa.
_:usa      a       :Location
_:usa      :name   "United States"
_:usa      :type   "country".
_:usa      :within _:namerica.
_:namerica a       :Location
_:namerica :name   "North America"
_:namerica :type   :"continent"
```

在这个例子中，图的顶点被写为：`_：someName`。这个名字并不意味着这个文件以外的任何东西。它的存在只是帮助我们明确哪些三元组引用了同一顶点。当谓语表示边时，该宾语是一个顶点，如 `_:idaho :within _:usa.` 。当谓语是一个属性时，该宾语是一个字符串，如 `_:usa :name "United States"`

一遍又一遍地重复相同的主语看起来相当重复，但幸运的是，可以使用分号来说明关于同一主语的多个事情。这使得Turtle格式相当不错，可读性强：参见例2-7。

例**2-7** 一种相对例**2-6**写入数据的更为简洁的方法。

```
@prefix : <urn:example:>.
_:lucy      a :Person;   :name "Lucy";          :bornIn _:idaho.
_:idaho     a :Location; :name "Idaho";         :type "state";   :within _:usa
_:usa       a :Loaction; :name "United States"; :type "country"; :within _:namerica.
_:namerica  a :Location; :name "North America"; :type "continent".
```

语义网络

如果你阅读更多关于三元组存储的信息，你可能会被卷入关于语义网络的文章漩涡中。三元组存储数据模型完全独立于语义网络，例如，Datomic【40】是三元组存储[vii]，并没有声称与它有任何关系。但是，由于在很多人眼中这两者紧密相连，我们应该简要地讨论一下。

> vii. 从技术上讲，Datomic使用的是五元组而不是三元组，两个额外的字段是用于版本控制的元数据 ↩

从本质上讲语义网是一个简单且合理的想法：网站已经将信息发布为文字和图片供人类阅读，为什么不将信息作为机器可读的数据也发布给计算机呢？资源描述框架（RDF）【41】的目的是作为不同网站以一致的格式发布数据的一种机制，允许来自不同网站的数据自动合并成一个数据网络 - 一种互联网范围内的“关于一切的数据库“。

不幸的是，这个语义网在二十一世纪初被过度使用，但到目前为止没有任何迹象表明已在实践中实现，这使得许多人呲之以鼻。它还遭受了过多的令人眼花缭乱的缩略词，过于复杂的标准提议和狂妄自大的苦果。

然而，如果仔细观察这些失败，语义Web项目还是拥有很多优秀的工作成果。即使你没有兴趣在语义网上发布RDF数据，三元组也可以成为应用程序的良好内部数据模型。

RDF数据模型

例2-7中使用的Turtle语言是一种用于RDF数据的人可读格式。有时候，RDF也可以以XML格式编写，不过完成同样的事情会相对啰嗦，参见例2-8。Turtle/N3是更可取的，因为它更容易阅读，像Apache Jena 【42】这样的工具可以根据需要在不同的RDF格式之间进行自动转换。

例**2-8** 用**RDF/XML**语法表示例**2-7**的数据

```
<rdf:RDF xmlns="urn:example:"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <Location rdf:nodeID="idaho">
        <name>Idaho</name>
        <type>state</type>
        <within>
            <Location rdf:nodeID="usa">
                <name>United States</name>
                <type>country</type>
                <within>
                    <Location rdf:nodeID="namerica">
                        <name>North America</name>
                        <type>continent</type>
                    </Location>
                </within>
            </Location>
        </within>
    </Location>
    <Person rdf:nodeID="lucy">
        <name>Lucy</name>
        <bornIn rdf:nodeID="idaho"/>
    </Person>
</rdf:RDF>
```

RDF有一些奇怪之处，因为它是为了在互联网上交换数据而设计的。三元组的主语，谓语和宾语通常是URI。例如，谓语可能是一个URI，如 `<http://my-company.com/namespace#within>` 或 `<http://my-company.com/namespace#lives_in>` ，而不仅仅是 `WITHIN` 或 `LIVES_IN` 。这个设计背后的原因为了让你能够把你的数据和其他人的数据结合起来，如果他们赋予单词 `within` 或者 `lives_in` 不同的含义，两者也不会冲突，因为它们的谓语实际上是 `<http://other.org/foo#within>` 和 `<http://other.org/foo#lives_in>` 。

从RDF的角度来看，URL `<http://my-company.com/namespace>` 不一定需要能解析成什么东西，它只是一个命名空间。为避免与 `http://URL` 混淆，本节中的示例使用不可解析的URI，如 `urn：example：within` 。幸运的是，你只需在文件顶部指定一个前缀，然后就不用再管了。

SPARQL查询语言

SPARQL是一种用于三元组存储的面向RDF数据模型的查询语言，【43】。（它是SPARQL协议和RDF查询语言的缩写，发音为“sparkle”。）SPARQL早于Cypher，并且由于Cypher的模式匹配借鉴于SPARQL，这使得它们看起来非常相似【37】。

与之前相同的查询 - 查找从美国转移到欧洲的人 - 使用SPARQL比使用Cypher甚至更为简洁（参见例2-9）。

例**2-9** 与示例**2-4**相同的查询，用**SPARQL**表示

```
PREFIX : <urn:example:>
SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn  / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}
```

结构非常相似。以下两个表达式是等价的（SPARQL中的变量以问号开头）：

```
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location)   # Cypher
?person :bornIn / :within* ?location.                   # SPARQL
```

因为RDF不区分属性和边，而只是将它们作为谓语，所以可以使用相同的语法来匹配属性。在下面的表达式中，变量 `usa` 被绑定到任意具有值为字符串 `"United States"` 的 `name` 属性的顶点：

```
(usa {name:'United States'})   # Cypher
?usa :name "United States".    # SPARQL
```

SPARQL是一种很好的查询语言——哪怕语义网从未实现，它仍然可以成为一种应用程序内部使用的强大工具。

> 图形数据库与网络模型相比较
>
> 在“文档数据库是否在重蹈覆辙？”中，我们讨论了CODASYL和关系模型如何竞相解决IMS中的多对多关系问题。乍一看，CODASYL的网络模型看起来与图模型相似。CODASYL是否是图形数据库的第二个变种？
>
> 不，他们在几个重要方面有所不同：
>
> - 在CODASYL中，数据库有一个模式，用于指定哪种记录类型可以嵌套在其他记录类型中。在图形数据库中，不存在这样的限制：任何顶点都可以具有到其他任何顶点的边。这为应用程序适应不断变化的需求提供了更大的灵活性。
> - 在CODASYL中，达到特定记录的唯一方法是遍历其中的一个访问路径。在图形数据库中，可以通过其唯一ID直接引用任何顶点，也可以使用索引来查找具有特定值的顶点。
> - 在CODASYL，记录的后续是一个有序集合，所以数据库的人不得不维持排序（这会影响存储布局），并且插入新记录到数据库的应用程序不得不担心的新记录在这些集合中的位置。在图形数据库中，顶点和边不是有序的（只能在查询时对结果进行排序）。
> - 在CODASYL中，所有查询都是命令式的，难以编写，并且很容易因架构中的变化而受到破坏。在图形数据库中，如果需要，可以在命令式代码中编写遍历，但大多数图形数据库也支持高级声明式查询语言，如Cypher或SPARQL。

基础：Datalog

Datalog是比SPARQL或Cypher更古老的语言，在20世纪80年代被学者广泛研究【44,45,46】。它在软件工程师中不太知名，但是它是重要的，因为它为以后的查询语言提供了基础。

在实践中，Datalog被用于少数的数据系统中：例如，它是Datomic 【40】的查询语言，Cascalog 【47】是一种用于查询Hadoop大数据集的Datalog实现[viii]。

> viii. Datomic和Cascalog使用Datalog的Clojure S表达式语法。在下面的例子中使用了一个更容易阅读的Prolog语法，但两者没有任何功能差异。↩

Datalog的数据模型类似于三元组模式，但进行了一点泛化。把三元组写成谓语（主语，宾语），而不是写三元语（主语，宾语，宾语）。例2-10显示了如何用Datalog写入我们的例子中的数据。

例**2-10** 用**Datalog**来表示图**2-5**中的数据子集

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).
```

既然已经定义了数据，我们可以像之前一样编写相同的查询，如例2-11所示。它看起来有点不同于Cypher或SPARQL的等价物，但是请不要放弃它。Datalog是Prolog的一个子集，如果你学过计算机科学，你可能已经见过。

例**2-11** 与示例**2-4**相同的查询，用**Datalog**表示

```
within_recursive(Location, Name) :- name(Location, Name). /* Rule 1 */

within_recursive(Location, Name) :- within(Location, Via), /* Rule 2 */
                                    within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name), /* Rule 3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc, BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc, LivingIn).

?- migrated(Who, 'United States', 'Europe'). /* Who = 'Lucy'. */
```

Cypher和SPARQL使用SELECT立即跳转，但是Datalog一次只进行一小步。我们定义规则，以将新谓语告诉数据库：在这里，我们定义了两个新的谓语，`_recursive` 和 `migrated` 。这些谓语不是存储在数据库中的三元组中，而是它们是从数据或其他规则派生而来的。规则可以引用其他规则，就像函数可以调用其他函数或者递归地调用自己一样。像这样，复杂的查询可以一次构建其中的一小块。

在规则中，以大写字母开头的单词是变量，谓语则用Cypher和SPARQL的方式一样来匹配。例如， `name(Location, Name)` 通过变量绑定 `Location = namerica` 和 `Name ='North America'` 可以匹配三元组 `name(namerica, 'North America')` 。

要是系统可以在 `:-` 操作符的右侧找到与所有谓语的一个匹配，就运用该规则。当规则运用时，就好像通过 `:-` 的左侧将其添加到数据库（将变量替换成它们匹配的值）。

因此，一种可能的应用规则的方式是：

1. 数据库存在 `name(namerica, 'North America')` ，故运用规则1。它生成 `within_recursive(namerica, 'North America')` 。
2. 数据库存在 `within(usa, namerica)` ，在上一步骤中生成 `within_recursive(namerica, 'North America')` ，故运用规则2。它会产生 `within_recursive(usa, 'North America')` 。
3. 数据库存在 `within(idaho, usa)` ，在上一步生成 `within_recursive(usa, 'North America')` ，故运用规则2。它产生 `within_recursive(idaho, 'North America')` 。

通过重复应用规则1和2， `within_recursive` 谓语可以告诉我们在数据库中包含北美（或任何其他位置名称）的所有位置。这个过程如图2-6所示。

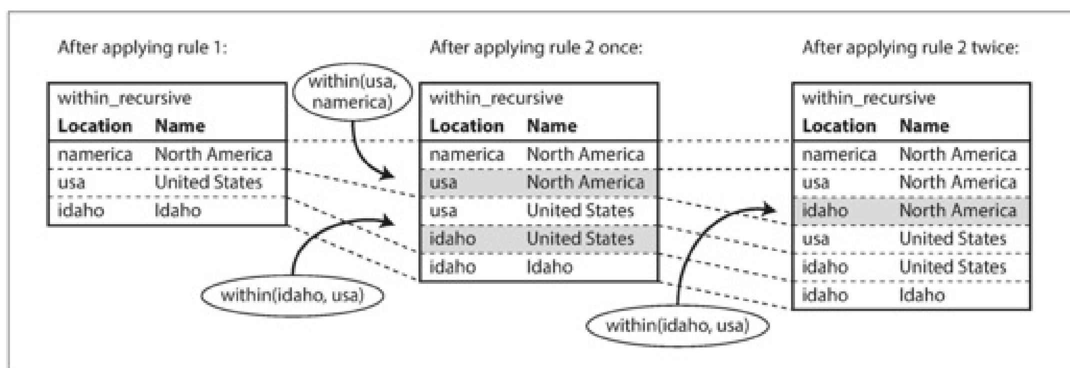


图**2-6** 使用示例**2-11**中的**Datalog**规则来确定爱达荷州在北美。

现在规则3可以找到出生在某个地方 `BornIn` 的人，并住在某个地方 `LivingIn` 。通过查询 `BornIn ='United States'` 和 `LivingIn ='Europe'` ，并将此人作为变量 `Who` ，让Datalog系统找出变量 `Who` 会出现哪些值。因此，最后得到了与早先的Cypher和SPARQL查询相同的答案。

相对于本章讨论的其他查询语言，我们需要采取不同的思维方式来思考Datalog方法，但这是一种非常强大的方法，因为规则可以在不同的查询中进行组合和重用。虽然对于简单的一次性查询，显得不太方便，但是它可以更好地处理数据很复杂的情况。

本章小结

数据模型是一个巨大的课题，在本章中，我们快速浏览了各种不同的模型。我们没有足够的空间来详细介绍每个模型的细节，但是希望这个概述足以激起你的兴趣，以更多地了解最适合你的应用需求的模型。

在历史上，数据最开始被表示为一棵大树（层次数据模型），但是这不利于表示多对多的关系，所以发明了关系模型来解决这个问题。最近，开发人员发现一些应用程序也不适合采用关系模型。新的非关系型“NoSQL”数据存储在两个主要方向上存在分歧：

1. 文档数据库的应用场景是：数据通常是自我包含的，而且文档之间的关系非常稀少。
2. 图形数据库用于相反的场景：任意事物都可能与任何事物相关联。

这三种模型（文档，关系和图形）在今天都被广泛使用，并且在各自的领域都发挥很好。一个模型可以用另一个模型来模拟 — 例如，图数据可以在关系数据库中表示 — 但结果往往是糟糕的。这就是为什么我们有着针对不同目的的不同系统，而不是一个单一的万能解决方案。

文档数据库和图数据库有一个共同点，那就是它们通常不会为存储的数据强制一个模式，这可以使应用程序更容易适应不断变化的需求。但是应用程序很可能仍会假定数据具有一定的结构；这只是模式是明确的（写入时强制）还是隐含的（读取时处理）的问题。

每个数据模型都具有各自的查询语言或框架，我们讨论了几个例子：SQL，MapReduce，MongoDB的聚合管道，Cypher，SPARQL和Datalog。我们也谈到了CSS和XSL/XPath，它们不是数据库查询语言，而包含有趣的相似之处。

虽然我们已经覆盖了很多层面，但仍然有许多数据模型没有提到。举几个简单的例子：

- 使用基因组数据的研究人员通常需要执行序列相似性搜索，这意味着需要一个很长的字符串（代表一个DNA分子），并在一个拥有类似但不完全相同的字符串的大型数据库中寻找匹配。这里所描述的数据库都不能处理这种用法，这就是为什么研究人员编写了像GenBank这样的专门的基因组数据库软件的原因【48】。
- 粒子物理学家数十年来一直在进行大数据类型的大规模数据分析，像大型强子对撞机（LHC）这样的项目现在可以工作在数百亿兆字节的范围内！在这样的规模下，需要定制解决方案来阻住硬件成本的失控【49】。
- 全文搜索可以说是一种经常与数据库一起使用的数据模型。信息检索是一个很大的专业课题，我们不会在本书中详细介绍，但是我们将在第三章和第三章中介绍搜索索引。

让我们暂时将其放在一边。在下一章中，我们将讨论在实现本章描述的数据模型时会遇到的一些权衡。

参考文献

1. Edgar F. Codd: “A Relational Model of Data for Large Shared Data Banks,” *Communications of the ACM*, volume 13, number 6, pages 377–387, June 1970. doi:10.1145/362384.362685

2. Michael Stonebraker and Joseph M. Hellerstein: "What Goes Around Comes Around," in *Readings in Database Systems*, 4th edition, MIT Press, pages 2–41, 2005. ISBN: 978-0-262-69314-1
3. Pramod J. Sadalage and Martin Fowler: *NoSQL Distilled*. Addison-Wesley, August 2012. ISBN: 978-0-321-82662-6
4. Eric Evans: "NoSQL: What's in a Name?," *blog.sym-link.com*, October 30, 2009.
5. James Phillips: "Surprises in Our NoSQL Adoption Survey," *blog.couchbase.com*, February 8, 2012.
6. Michael Wagner: *SQL/XML:2006 – Evaluierung der Standardkonformität ausgewählter Datenbanksysteme*. Diplomica Verlag, Hamburg, 2010. ISBN: 978-3-836-64609-3
7. "XML Data in SQL Server," SQL Server 2012 documentation, *technet.microsoft.com*, 2013.
8. "PostgreSQL 9.3.1 Documentation," The PostgreSQL Global Development Group, 2013.
9. "The MongoDB 2.4 Manual," MongoDB, Inc., 2013.
10. "RethinkDB 1.11 Documentation," *rethinkdb.com*, 2013.
11. "Apache CouchDB 1.6 Documentation," *docs.couchdb.org*, 2014.
12. Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: "On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.
13. Rick Long, Mark Harrington, Robert Hain, and Geoff Nicholls: *IMS Primer*. IBM Redbook SG24-5352-00, IBM International Technical Support Organization, January 2000.
14. Stephen D. Bartlett: "IBM's IMS—Myths, Realities, and Opportunities," The Clipper Group Navigator, TCG2013015LI, July 2013.
15. Sarah Mei: "Why You Should Never Use MongoDB," *sarahmei.com*, November 11, 2013.
16. J. S. Knowles and D. M. R. Bell: "The CODASYL Model," in *Databases—Role and Structure: An Advanced Course*, edited by P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, pages 19–56, Cambridge University Press, 1984. ISBN: 978-0-521-25430-4
17. Charles W. Bachman: "The Programmer as Navigator," *Communications of the ACM*, volume 16, number 11, pages 653–658, November 1973. doi:10.1145/355611.362534

18. Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: “Architecture of a Database System,” *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007. doi:10.1561/1900000002

19. Sandeep Parikh and Kelly Stirman: “Schema Design for Time Series Data in MongoDB,” *blog.mongodb.org*, October 30, 2013.

20. Martin Fowler: “Schemaless Data Structures,” *martinfowler.com*, January 7, 2013.

21. Amr Awadallah: “Schema-on-Read vs. Schema-on-Write,” at *Berkeley EECS RAD Lab Retreat*, Santa Cruz, CA, May 2009.

22. Martin Odersky: “The Trouble with Types,” at *Strange Loop*, September 2013.

23. Conrad Irwin: “MongoDB—Confessions of a PostgreSQL Lover,” at *HTML5DevConf*, October 2013.

24. “Percona Toolkit Documentation: pt-online-schema-change,” Percona Ireland Ltd., 2013.

25. Rany Keddo, Tobias Bielohlawek, and Tobias Schmidt: “Large Hadron Migrator,” SoundCloud, 2013. Shlomi Noach:

    “gh-ost: GitHub's Online Schema Migration Tool for MySQL,” *githubengineering.com*, August 1, 2016.

26. James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: “Spanner: Google’s Globally-Distributed Database,” at *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

27. Donald K. Burleson: “Reduce I/O with Oracle Cluster Tables,” *dba-oracle.com*.

28. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: “Bigtable: A Distributed Storage System for Structured Data,” at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

29. Bobbie J. Cochrane and Kathy A. McKnight: “DB2 JSON Capabilities, Part 1: Introduction to DB2 JSON,” IBM developerWorks, June 20, 2013.

30. Herb Sutter: “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,” *Dr. Dobb's Journal*, volume 30, number 3, pages 202-210, March 2005.

31. Joseph M. Hellerstein: “The Declarative Imperative: Experiences and Conjectures in Distributed Logic,” Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech report UCB/EECS-2010-90, June 2010.

32. Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters," at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

33. Craig Kerstiens: "JavaScript in Your Postgres," *blog.heroku.com*, June 5, 2013.

34. Nathan Bronson, Zach Amsden, George Cabrera, et al.: "TAO: Facebook's Distributed Data Store for the Social Graph," at *USENIX Annual Technical Conference* (USENIX ATC), June 2013.

35. "Apache TinkerPop3.2.3 Documentation," *tinkerpop.apache.org*, October 2016.

36. "The Neo4j Manual v2.0.0," Neo Technology, 2013. Emil Eifrem: Twitter correspondence, January 3, 2014.

37. David Beckett and Tim Berners-Lee: "Turtle – Terse RDF Triple Language," W3C Team Submission, March 28, 2011.

38. "Datomic Development Resources," Metadata Partners, LLC, 2013. W3C RDF Working Group: "Resource Description Framework (RDF)," *w3.org*, 10 February 2004.

39. "Apache Jena," Apache Software Foundation.

40. Steve Harris, Andy Seaborne, and Eric Prud'hommeaux: "SPARQL 1.1 Query Language," W3C Recommendation, March 2013.

41. Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou: "Datalog and Recursive Query Processing," *Foundations and Trends in Databases*, volume 5, number 2, pages 105–195, November 2013. doi:10.1561/1900000017

42. Stefano Ceri, Georg Gottlob, and Letizia Tanca: "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Transactions on Knowledge and Data Engineering*, volume 1, number 1, pages 146–166, March 1989. doi:10.1109/69.43410

43. Serge Abiteboul, Richard Hull, and Victor Vianu: *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 978-0-201-53771-0, available online at *webdam.inria.fr/Alice*

44. Nathan Marz: "Cascalog," *cascalog.org*. Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, et al.:

    "GenBank," *Nucleic Acids Research*, volume 36, Database issue, pages D25–D30, December 2007. doi:10.1093/nar/gkm929

45. Fons Rademakers: "ROOT for Big Data Analysis," at *Workshop on the Future of Big Data Management*, London, UK, June 2013.

上一章	目录	下一章
第一章：可靠、可扩展、可维护	设计数据密集型应用	第三章：存储与检索

3. 存储与检索



> 建立秩序，省却搜索
>
> ——德国谚语

[TOC]

一个数据库在最基础的层次上需要完成两件事情：当你把数据交给数据库时，它应当把数据存储起来；而后当你向数据库要数据时，它应当把数据返回给你。

在第2章中，我们讨论了数据模型和查询语言，即程序员将数据录入数据库的格式，以及再次要回数据的机制。在本章中我们会从数据库的视角来讨论同样的问题：数据库如何存储我们提供的数据，以及如何在我们需要时重新找到数据。

作为程序员，为什么要关心数据库内部存储与检索的机理？你可能不会去从头开始实现自己的存储引擎，但是你确实需要从许多可用的存储引擎中选择一个合适的。而且为了调谐存储引擎以适配应用工作负载，你也需要大致了解存储引擎在底层究竟做什么。

特别需要注意，针对事务性负载和分析性负载优化的存储引擎之间存在巨大差异。稍后我们将在“事务处理还是分析？”一节中探讨这一区别，并在“列存储”中讨论一系列针对分析优化存储引擎。

但是，我们将从您最可能熟悉的两大类数据库：传统关系型数据库与很多所谓的“NoSQL”数据库开始，通过介绍它们的存储引擎来开始本章的内容。我们会研究两大类存储引擎：日志结构（**log-structured**）的存储引擎，以及面向页面（**page-oriented**）的存储引擎（例如B树）。

驱动数据库的数据结构

世界上最简单的数据库可以用两个Bash函数实现：

```
#!/bin/bash
db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

这两个函数实现了键值存储的功能。执行 `db_set key value` ，会将 键（**key**）和值（**value**） 存储在数据库中。键和值（几乎）可以是你喜欢的任何东西，例如，值可以是JSON文档。然后调用 `db_get key` ，查找与该键关联的最新值并将其返回。

麻雀虽小，五脏俱全：

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}' $

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

底层的存储格式非常简单：一个文本文件，每行包含一条逗号分隔的键值对（忽略转义问题的话，大致与CSV文件类似）。每次对 `db_set` 的调用都会向文件末尾追加记录，所以更新键的时候旧版本的值不会被覆盖 —— 因而查找最新值的时候，需要找到文件中键最后一次出现的位置（因此 `db_get` 中使用了 `tail -n 1` 。)

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

`db_set` 函数对于极其简单的场景其实有非常好的性能，因为在文件尾部追加写入通常是非常高效的。与 `db_set` 做的事情类似，许多数据库在内部使用了日志（**log**），也就是一个仅追加（**append-only**）的数据文件。真正的数据库有更多的问题需要处理（如并发控制，回收

磁盘空间以避免日志无限增长，处理错误与部分写入的记录），但基本原理是一样的。日志极其有用，我们还将在本书的其它部分重复见到它好几次。

> 日志（**log**）这个词通常指应用日志：即应用程序输出的描述发生事情的文本。本书在更普遍的意义下使用日志这一词：一个仅追加的记录序列。它可能压根就不是给人类看的，使用二进制格式，并仅能由其他程序读取。

另一方面，如果这个数据库中有着大量记录，则这个 `db_get` 函数的性能会非常糟糕。每次你想查找一个键时， `db_get` 必须从头到尾扫描整个数据库文件来查找键的出现。用算法的语言来说，查找的开销是 `O(n)` ：如果数据库记录数量 n 翻了一倍，查找时间也要翻一倍。这就不好了。

为了高效查找数据库中特定键的值，我们需要一个数据结构：索引（**index**）。本章将介绍一系列的索引结构，并它们进行对比。索引背后的大致思想是，保存一些额外的元数据作为路标，帮助你找到想要的数据。如果您想在同一份数据中以几种不同的方式进行搜索，那么你也许需要不同的索引，建在数据的不同部分上。

索引是从主数据衍生的附加（**additional**）结构。许多数据库允许添加与删除索引，这不会影响数据的内容，它只影响查询的性能。维护额外的结构会产生开销，特别是在写入时。写入性能很难超过简单地追加写入文件，因为追加写入是最简单的写入操作。任何类型的索引通常都会减慢写入速度，因为每次写入数据时都需要更新索引。

这是存储系统中一个重要的权衡：精心选择的索引加快了读查询的速度，但是每个索引都会拖慢写入速度。因为这个原因，数据库默认并不会索引所有的内容，而需要你（程序员或DBA）通过对应用查询模式的了解来手动选择索引。你可以选择能为应用带来最大收益，同时又不会引入超出必要开销的索引。

哈希索引

让我们从键值数据（**key-value Data**）的索引开始。这不是您可以索引的唯一数据类型，但键值数据是很常见的。对于更复杂的索引来说，这是一个有用的构建模块。

键值存储与在大多数编程语言中可以找到的字典（**dictionary**）类型非常相似，通常字典都是用散列映射（**hash map**）（或哈希表（**hash table**））实现的。哈希映射在许多算法教科书中都有描述【1,2】，所以这里我们不会讨论它的工作细节。既然我们已经有内存中数据结构——哈希映射，为什么不使用它来索引在磁盘上的数据呢？

假设我们的数据存储只是一个追加写入的文件，就像前面的例子一样。那么最简单的索引策略就是：保留一个内存中的哈希映射，其中每个键都映射到一个数据文件中的字节偏移量，指明了可以找到对应值的位置，如图3-1所示。当你将新的键值对追加写入文件中时，还要更新散列映射，以反映刚刚写入的数据的偏移量（这同时适用于插入新键与更新现有键）。当你想查找一个值时，使用哈希映射来查找数据文件中的偏移量，寻找（**seek**）该位置并读取该值。

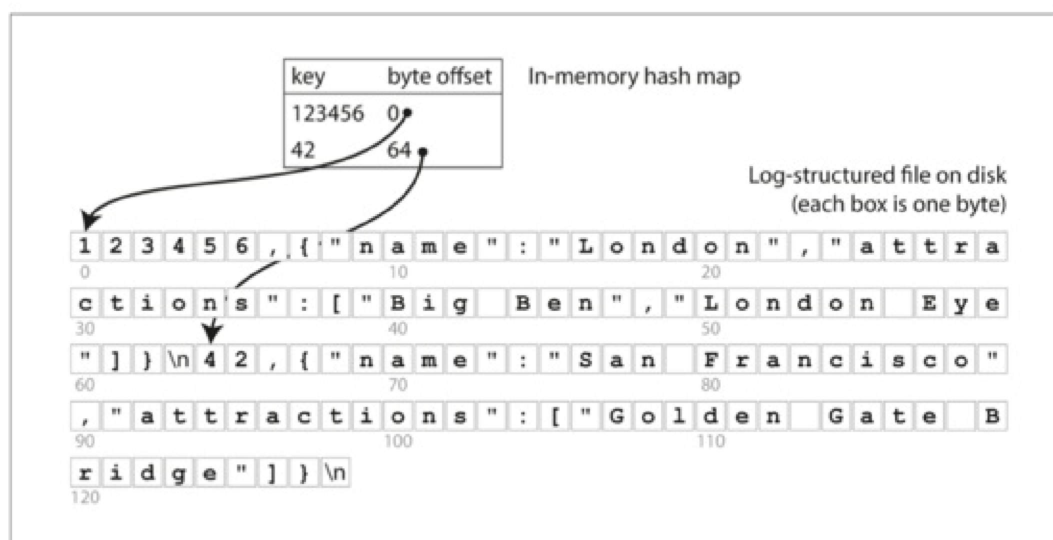


图**3-1** 以类**CSV**格式存储键值对的日志，并使用内存哈希映射进行索引。

听上去简单，但这是一个可行的方法。现实中，Bitcask实际上就是这么做的（Riak中默认的存储引擎）【3】。Bitcask提供高性能的读取和写入操作，但所有键必须能放入可用内存中，因为哈希映射完全保留在内存中。这些值可以使用比可用内存更多的空间，因为可以从磁盘上通过一次 `seek` 加载所需部分，如果数据文件的那部分已经在文件系统缓存中，则读取根本不需要任何磁盘I/O。

像Bitcask这样的存储引擎非常适合每个键的值经常更新的情况。例如，键可能是视频的URL，值可能是它播放的次数（每次有人点击播放按钮时递增）。在这种类型的工作负载中，有很多写操作，但是没有太多不同的键——每个键有很多的写操作，但是将所有键保存在内存中是可行的。

直到现在，我们只是追加写入一个文件 —— 所以如何避免最终用完磁盘空间？一种好的解决方案是，将日志分为特定大小的段，当日志增长到特定尺寸时关闭当前段文件，并开始写入一个新的段文件。然后，我们就可以对这些段进行压缩（**compaction**），如图3-2所示。压缩意味着在日志中丢弃重复的键，只保留每个键的最近更新。

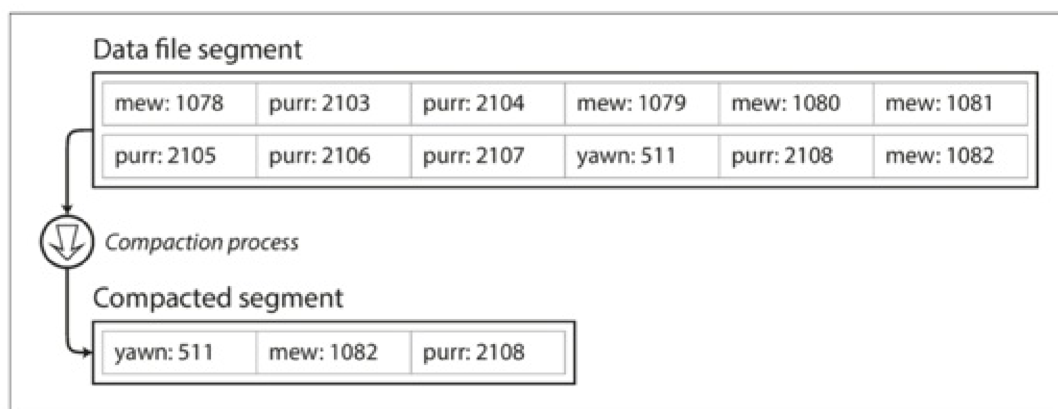


图**3-2** 压缩键值更新日志（统计猫视频的播放次数），只保留每个键的最近值

而且，由于压缩经常会使得段变得很小（假设在一个段内键被平均重写了好几次），我们也可以在执行压缩的同时将多个段合并在一起，如图3-3所示。段被写入后永远不会被修改，所以合并的段被写入一个新的文件。冻结段的合并和压缩可以在后台线程中完成，在进行时，我们仍然可以继续使用旧的段文件来正常提供读写请求。合并过程完成后，我们将读取请求转换为使用新的合并段而不是旧段 —— 然后可以简单地删除旧的段文件。

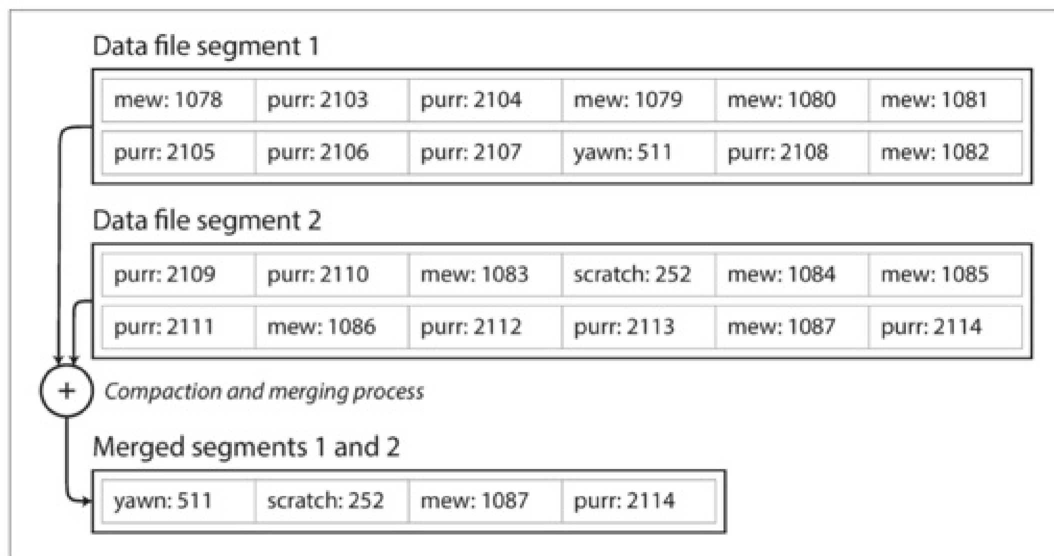


图**3-3** 同时执行压缩和分段合并

每个段现在都有自己的内存散列表，将键映射到文件偏移量。为了找到一个键的值，我们首先检查最近段的哈希映射;如果键不存在，我们检查第二个最近的段，依此类推。合并过程保持细分的数量，所以查找不需要检查许多哈希映射。 大量的细节进入实践这个简单的想法工作。简而言之，一些真正实施中重要的问题是：

文件格式

CSV不是日志的最佳格式。使用二进制格式更快，更简单，首先以字节为单位对字符串的长度进行编码，然后使用原始字符串（不需要转义）。

删除记录

如果要删除一个键及其关联的值，则必须在数据文件（有时称为逻辑删除）中附加一个特殊的删除记录。当日志段被合并时，逻辑删除告诉合并过程放弃删除键的任何以前的值。

崩溃恢复

如果数据库重新启动，则内存散列映射将丢失。原则上，您可以通过从头到尾读取整个段文件并在每次按键时注意每个键的最近值的偏移量来恢复每个段的哈希映射。但是，如果段文件很大，这可能需要很长时间，这将使服务器重新启动痛苦。 Bitcask通过存储加速恢复磁盘上每个段的哈希映射的快照，可以更快地加载到内存中。

部分写入记录

数据库可能随时崩溃，包括将记录附加到日志中途。 Bitcask文件包含校验和，允许检测和忽略日志的这些损坏部分。

并发控制

由于写操作是以严格顺序的顺序附加到日志中的，所以常见的实现选择是只有一个写入器线程。数据文件段是附加的，否则是不可变的，所以它们可以被多个线程同时读取。

乍一看，只有追加日志看起来很浪费：为什么不更新文件，用新值覆盖旧值？但是只能追加设计的原因有几个：

- 追加和分段合并是顺序写入操作，通常比随机写入快得多，尤其是在磁盘旋转硬盘上。在某种程度上，顺序写入在基于闪存的固态硬盘（**SSD**）上也是优选的【4】。我们将在第83页的“比较B-树和LSM-树”中进一步讨论这个问题。
- 如果段文件是附加的或不可变的，并发和崩溃恢复就简单多了。例如，您不必担心在覆盖值时发生崩溃的情况，而将包含旧值和新值的一部分的文件保留在一起。
- 合并旧段可以避免数据文件随着时间的推移而分散的问题。

但是，哈希表索引也有局限性：

- 散列表必须能放进内存

  如果你有非常多的键，那真是倒霉。原则上可以在磁盘上保留一个哈希映射，不幸的是磁盘哈希映射很难表现优秀。它需要大量的随机访问I/O，当它变满时增长是很昂贵的，并且散列冲突需要很多的逻辑【5】。

- 范围查询效率不高。例如，您无法轻松扫描kitty00000和kitty99999之间的所有键——您必须在散列映射中单独查找每个键。

在下一节中，我们将看看一个没有这些限制的索引结构。

SSTables和LSM树

在图3-3中，每个日志结构存储段都是一系列键值对。这些对按照它们写入的顺序出现，日志中稍后的值优先于日志中较早的相同键的值。除此之外，文件中键值对的顺序并不重要。

现在我们可以对段文件的格式做一个简单的改变：我们要求键值对的序列按键排序。乍一看，这个要求似乎打破了我们使用顺序写入的能力，但是我们马上就会明白这一点。

我们把这个格式称为排序字符串表（**Sorted String Table**），简称SSTable。我们还要求每个键只在每个合并的段文件中出现一次（压缩过程已经保证）。与使用散列索引的日志段相比，SSTable有几个很大的优势：

1. 合并段是简单而高效的，即使文件大于可用内存。这种方法就像归并排序算法中使用的方法一样，如图3-4所示：您开始并排读取输入文件，查看每个文件中的第一个键，复制最低键（根据排序顺序）到输出文件，并重复。这产生一个新的合并段文件，也按键排

序。

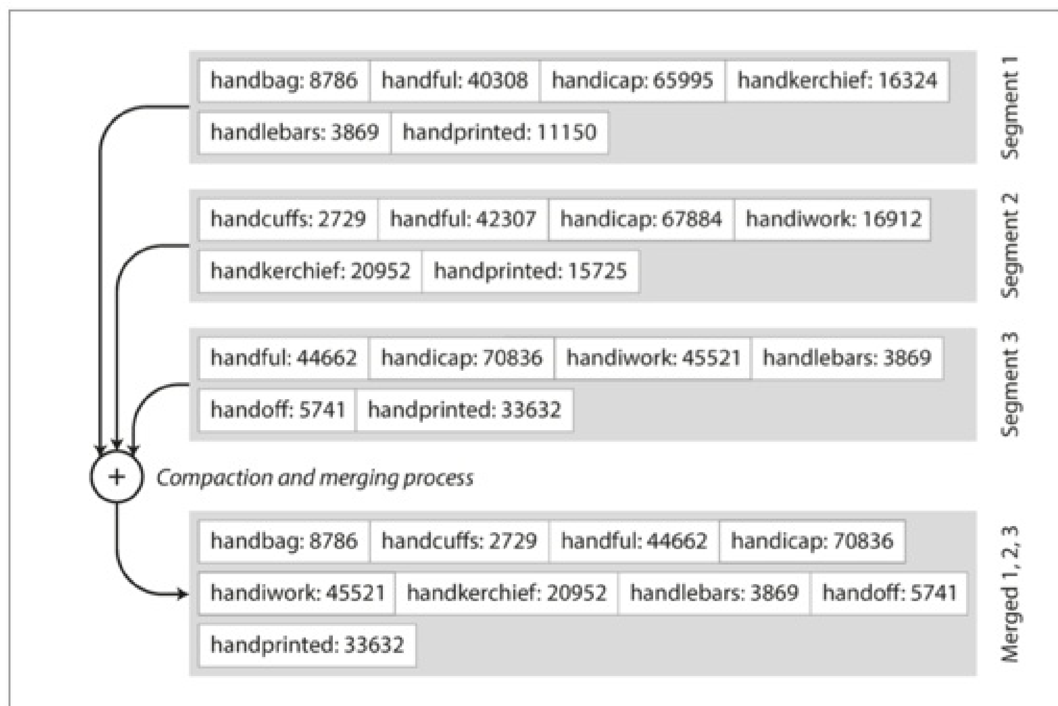


图**3-4** 合并几个**SSTable**段，只保留每个键的最新值

如果在几个输入段中出现相同的键，该怎么办？请记住，每个段都包含在一段时间内写入数据库的所有值。这意味着一个输入段中的所有值必须比另一个段中的所有值更新（假设我们总是合并相邻的段）。当多个段包含相同的键时，我们可以保留最近段的值，并丢弃旧段中的值。

2. 为了在文件中找到一个特定的键，你不再需要保存内存中所有键的索引。以图3-5为例：假设你正在内存中寻找键 `handiwork` ，但是你不知道段文件中该关键字的确切偏移量。然而，你知道 `handbag` 和 `handsome` 的偏移，而且由于排序特性，你知道 `handiwork` 必须出现在这两者之间。这意味着您可以跳到 `handbag` 的偏移位置并从那里扫描，直到您找到 `handiwork` （或没找到，如果该文件中没有该键）。

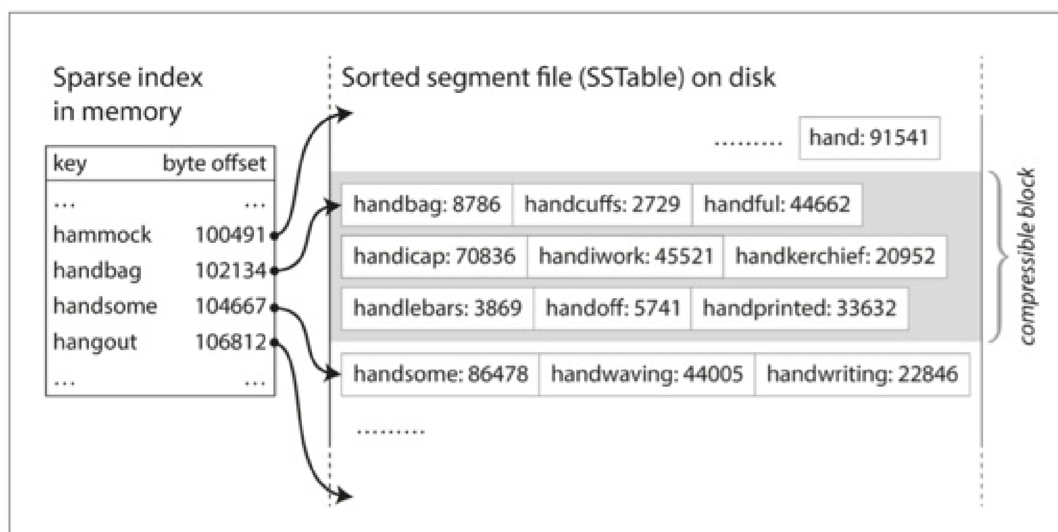


图3-5 具有内存索引的SSTable

您仍然需要一个内存中索引来告诉您一些键的偏移量，但它可能很稀疏：每几千字节的段文件就有一个键就足够了，因为几千字节可以很快被扫描[i]。

1. 由于读取请求无论如何都需要扫描所请求范围内的多个键值对，因此可以将这些记录分组到块中，并在将其写入磁盘之前对其进行压缩（如图3-5中的阴影区域所示）。稀疏内存中索引的每个条目都指向压缩块的开始处。除了节省磁盘空间之外，压缩还可以减少IO带宽的使用。

> i. 如果所有的键与值都是定长的，你可以使用段文件上的二分查找并完全避免使用内存索引。然而实践中键值通常都是变长的，因此如果没有索引，就很难知道记录的分界点（前一条记录结束，后一条记录开始的地方）↩

构建和维护SSTables

到目前为止，但是如何让你的数据首先被按键排序呢？我们的传入写入可以以任何顺序发生。

在磁盘上维护有序结构是可能的（参阅“B树”），但在内存保存则要容易得多。有许多可以使用的众所周知的树形数据结构，例如红黑树或AVL树【2】。使用这些数据结构，您可以按任何顺序插入键，并按排序顺序读取它们。

现在我们可以使我们的存储引擎工作如下：

- 写入时，将其添加到内存中的平衡树数据结构（例如，红黑树）。这个内存树有时被称为内存表（**memtable**）。
- 当内存表大于某个阈值（通常为几兆字节）时，将其作为SSTable文件写入磁盘。这可以高效地完成，因为树已经维护了按键排序的键值对。新的SSTable文件成为数据库的最新部分。当SSTable被写入磁盘时，写入可以继续到一个新的内存表实例。
- 为了提供读取请求，首先尝试在内存表中找到关键字，然后在最近的磁盘段中，然后在下一个较旧的段中找到该关键字。
- 有时会在后台运行合并和压缩过程以组合段文件并丢弃覆盖或删除的值。

这个方案效果很好。它只会遇到一个问题：如果数据库崩溃，则最近的写入（在内存表中，但尚未写入磁盘）将丢失。为了避免这个问题，我们可以在磁盘上保存一个单独的日志，每个写入都会立即被附加到磁盘上，就像在前一节中一样。该日志不是按排序顺序，但这并不重要，因为它的唯一目的是在崩溃后恢复内存表。每当内存表写出到SSTable时，相应的日志都可以被丢弃。

用SSTables制作LSM树

这里描述的算法本质上是LevelDB【6】和RocksDB【7】中使用的关键值存储引擎库，被设计嵌入到其他应用程序中。除此之外，LevelDB可以在Riak中用作Bitcask的替代品。在Cassandra和HBase中使用了类似的存储引擎【8】，这两种引擎都受到了Google的Bigtable

文档【9】（引入了SSTable和memtable）的启发。

最初这种索引结构是由Patrick O'Neil等人描述的。在日志结构合并树（或LSM树）【10】的基础上，建立在以前的工作上日志结构的文件系统【11】。基于这种合并和压缩排序文件原理的存储引擎通常被称为LSM存储引擎。

Lucene是Elasticsearch和Solr使用的一种全文搜索的索引引擎，它使用类似的方法来存储它的词典【12,13】。全文索引比键值索引复杂得多，但是基于类似的想法：在搜索查询中给出一个单词，找到提及单词的所有文档（网页，产品描述等）。这是通过键值结构实现的，其中键是单词（关键词（**term**）），值是包含单词（文章列表）的所有文档的ID的列表。在Lucene中，从术语到发布列表的这种映射保存在SSTable类的有序文件中，根据需要在后台合并【14】。

性能优化

与往常一样，大量的细节使得存储引擎在实践中表现良好。例如，当查找数据库中不存在的键时，LSM树算法可能会很慢：您必须检查内存表，然后将这些段一直回到最老的（可能必须从磁盘读取每一个），然后才能确定键不存在。为了优化这种访问，存储引擎通常使用额外的Bloom过滤器【15】。（布隆过滤器是用于近似集合内容的内存高效数据结构，它可以告诉您数据库中是否出现键，从而为不存在的键节省许多不必要的磁盘读取操作。

还有不同的策略来确定SSTables如何被压缩和合并的顺序和时间。最常见的选择是大小分层压实。 LevelDB和RocksDB使用平坦压缩（LevelDB因此得名），HBase使用大小分层，Cassandra同时支持【16】。在规模级别的调整中，更新和更小的SSTables先后被合并到更老的和更大的SSTable中。在水平压实中，关键范围被拆分成更小的SSTables，而较旧的数据被移动到单独的“水平”，这使得压缩能够更加递增地进行，并且使用更少的磁盘空间。

即使有许多微妙的东西，LSM树的基本思想 —— 保存一系列在后台合并的SSTables —— 简单而有效。即使数据集比可用内存大得多，它仍能继续正常工作。由于数据按排序顺序存储，因此可以高效地执行范围查询（扫描所有高于某些最小值和最高值的所有键），并且因为磁盘写入是连续的，所以LSM树可以支持非常高的写入吞吐量。

B树

刚才讨论的日志结构索引正处在逐渐被接受的阶段，但它们并不是最常见的索引类型。使用最广泛的索引结构在1970年被引入【17】，不到10年后变得“无处不在”【18】，B树经受了时间的考验。在几乎所有的关系数据库中，它们仍然是标准的索引实现，许多非关系数据库也使用它们。

像SSTables一样，B树保持按键排序的键值对，这允许高效的键值查找和范围查询。但这就是相似之处的结尾：B树有着非常不同的设计理念。

我们前面看到的日志结构索引将数据库分解为可变大小的段，通常是几兆字节或更大的大小，并且总是按顺序编写段。相比之下，B树将数据库分解成固定大小的块或页面，传统上大小为4KB（有时会更大），并且一次只能读取或写入一个页面。这种设计更接近于底层硬件，因为磁盘也被安排在固定大小的块中。

每个页面都可以使用地址或位置来标识，这允许一个页面引用另一个页面 —— 类似于指针，但在磁盘而不是在内存中。我们可以使用这些页面引用来构建一个页面树，如图3-6所示。

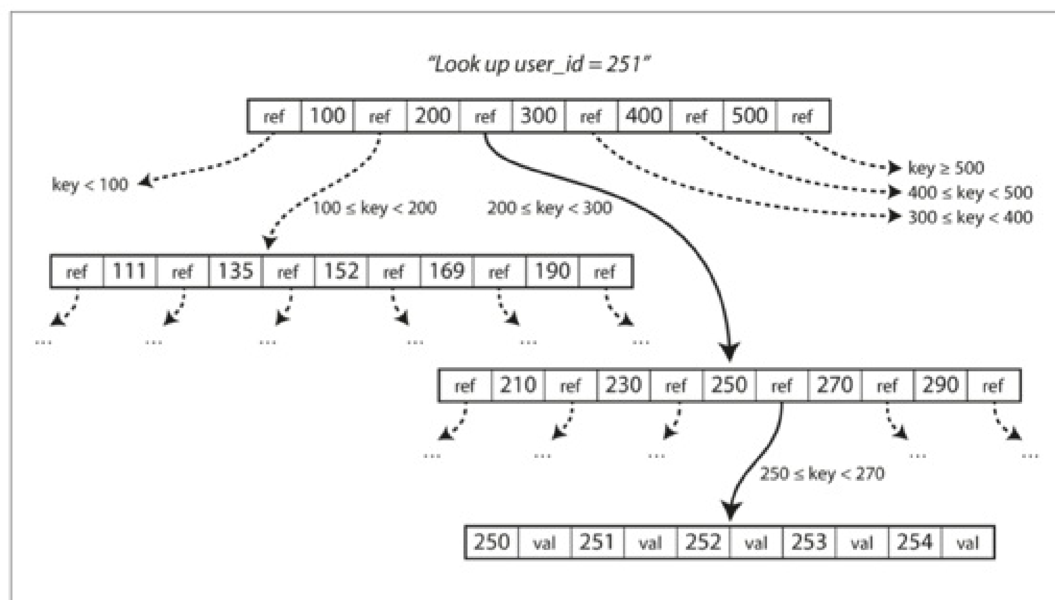


图**3-6** 使用**B**树索引查找一个键

一个页面会被指定为B树的根；在索引中查找一个键时，就从这里开始。该页面包含几个键和对子页面的引用。每个子页面负责一段连续范围的键，引用之间的键，指明了引用子页面的键范围。

在图3-6的例子中，我们正在寻找关键字 251 ，所以我们知道我们需要遵循边界 200 和 300 之间的页面引用。这将我们带到一个类似的页面，进一步打破了200 - 300到子范围。

最后，我们可以看到包含单个键（叶页）的页面，该页面包含每个键的内联值，或者包含对可以找到值的页面的引用。

在B树的一个页面中对子页面的引用的数量称为分支因子。例如，在图3-6中，分支因子是 6 。在实践中，分支因子取决于存储页面参考和范围边界所需的空间量，但通常是几百个。

如果要更新B树中现有键的值，则搜索包含该键的叶页，更改该页中的值，并将该页写回到磁盘（对该页的任何引用保持有效） 。如果你想添加一个新的键，你需要找到其范围包含新键的页面，并将其添加到该页面。如果页面中没有足够的可用空间容纳新键，则将其分成两个半满页面，并更新父页面以解释键范围的新分区，如图3-7所示[ii]。

> ii. 向B树中插入一个新的键是相当符合直觉的，但删除一个键（同时保持树平衡）就会牵扯很多其他东西了。↩

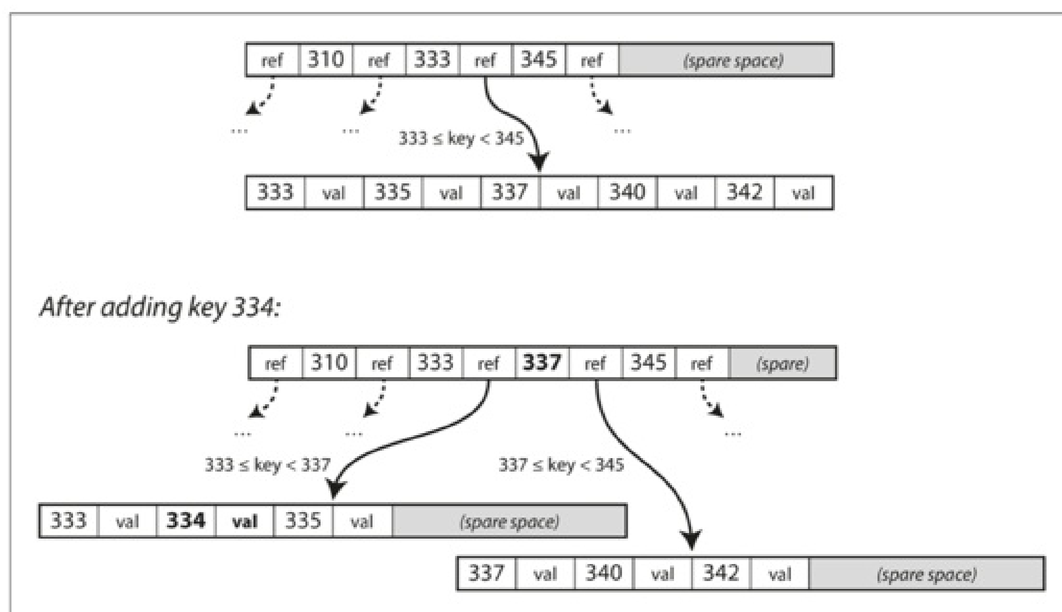


图**3-7** 通过分割页面来生长**B**树

该算法确保树保持平衡：具有 n 个键的B树总是具有 $O(log n)$ 的深度。大多数数据库可以放入一个三到四层的B树，所以你不需要遵追踪多页面引用来找到你正在查找的页面。（分支因子为 500 的 4KB 页面的四级树可以存储多达 256TB 。）

让B树更可靠

B树的基本底层写操作是用新数据覆盖磁盘上的页面。假定覆盖不改变页面的位置;即，当页面被覆盖时，对该页面的所有引用保持完整。这与日志结构索引（如LSM树）形成鲜明对比，后者只附加到文件（并最终删除过时的文件），但从不修改文件。

您可以考虑将硬盘上的页面覆盖为实际的硬件操作。在磁性硬盘驱动器上，这意味着将磁头移动到正确的位置，等待旋转盘上的正确位置出现，然后用新的数据覆盖适当的扇区。在固态硬盘上，由于SSD必须一次擦除和重写相当大的存储芯片块，所以会发生更复杂的事情【19】。

而且，一些操作需要覆盖几个不同的页面。例如，如果因为插入导致页面过度而拆分页面，则需要编写已拆分的两个页面，并覆盖其父页面以更新对两个子页面的引用。这是一个危险的操作，因为如果数据库在仅有一些页面被写入后崩溃，那么最终将导致一个损坏的索引（例如，可能有一个孤儿页面不是任何父项的子项）。

为了使数据库对崩溃具有韧性，B树实现通常会带有一个额外的磁盘数据结构：预写式日志（**WAL, write-ahead-log**）（也称为重做日志（**redo log**））。这是一个仅追加的文件，每个B树修改都可以应用到树本身的页面上。当数据库在崩溃后恢复时，这个日志被用来使B树恢复到一致的状态【5,20】。

更新页面的一个额外的复杂情况是，如果多个线程要同时访问B树，则需要仔细的并发控制 —— 否则线程可能会看到树处于不一致的状态。这通常通过使用锁存器（**latches**）（轻量级锁）保护树的数据结构来完成。日志结构化的方法在这方面更简单，因为它们在后台进行所

有的合并，而不会干扰传入的查询，并且不时地将旧的分段原子交换为新的分段。

B树优化

由于B树已经存在了这么久，许多优化已经发展了多年，这并不奇怪。仅举几例：

* 一些数据库（如LMDB）使用写时复制方案【21】，而不是覆盖页面并维护WAL进行崩溃恢复。修改的页面被写入到不同的位置，并且树中的父页面的新版本被创建，指向新的位置。这种方法对于并发控制也很有用，我们将在“快照隔离和可重复读”中看到。
* 我们可以通过不存储整个键来节省页面空间，但可以缩小它的大小。特别是在树内部的页面上，键只需要提供足够的信息来充当键范围之间的边界。在页面中包含更多的键允许树具有更高的分支因子，因此更少的层次
* 通常，页面可以放置在磁盘上的任何位置；没有什么要求附近的键范围页面附近的磁盘上。如果查询需要按照排序顺序扫描大部分关键字范围，那么每个页面的布局可能会非常不方便，因为每个读取的页面都可能需要磁盘查找。因此，许多B树实现尝试布局树，使得叶子页面按顺序出现在磁盘上。但是，随着树的增长，维持这个顺序是很困难的。相比之下，由于LSM树在合并过程中一次又一次地重写存储的大部分，所以它们更容易使顺序键在磁盘上彼此靠近。
* 额外的指针已添加到树中。例如，每个叶子页面可以在左边和右边具有对其兄弟页面的引用，这允许不跳回父页面就能顺序扫描。
* B树的变体如分形树【22】借用一些日志结构的思想来减少磁盘寻道（而且它们与分形无关）。

比较B树和LSM树

尽管B树实现通常比LSM树实现更成熟，但LSM树由于其性能特点也非常有趣。根据经验，通常LSM树的写入速度更快，而B树的读取速度更快【23】。 LSM树上的读取通常比较慢，因为它们必须在压缩的不同阶段检查几个不同的数据结构和SSTables。

然而，基准通常对工作量的细节不确定和敏感。 您需要测试具有特定工作负载的系统，以便进行有效的比较。 在本节中，我们将简要讨论一些在衡量存储引擎性能时值得考虑的事情。

LSM树的优点

B树索引必须至少两次写入每一段数据：一次写入预先写入日志，一次写入树页面本身（也许再次分页）。即使在该页面中只有几个字节发生了变化，也需要一次编写整个页面的开销。有些存储引擎甚至会覆盖同一个页面两次，以免在电源故障的情况下导致页面部分更新【24,25】。

由于反复压缩和合并SSTables，日志结构索引也会重写数据。这种影响 —— 在数据库的生命周期中写入数据库导致对磁盘的多次写入 —— 被称为写放大（**write amplification**）。需要特别关注的是固态硬盘，固态硬盘在磨损之前只能覆写一段时间。

在写入繁重的应用程序中，性能瓶颈可能是数据库可以写入磁盘的速度。在这种情况下，写放大会导致直接的性能代价：存储引擎写入磁盘的次数越多，可用磁盘带宽内的每秒写入次数越少。

而且，LSM树通常能够比B树支持更高的写入吞吐量，部分原因是它们有时具有较低的写放大（尽管这取决于存储引擎配置和工作负载），部分是因为它们顺序地写入紧凑的SSTable文件而不是必须覆盖树中的几个页面【26】。这种差异在磁性硬盘驱动器上尤其重要，顺序写入比随机写入快得多。

LSM树可以被压缩得更好，因此经常比B树在磁盘上产生更小的文件。 B树存储引擎会由于分割而留下一些未使用的磁盘空间：当页面被拆分或某行不能放入现有页面时，页面中的某些空间仍未被使用。由于LSM树不是面向页面的，并且定期重写SSTables以去除碎片，所以它们具有较低的存储开销，特别是当使用平坦压缩时【27】。

在许多固态硬盘上，固件内部使用日志结构化算法，将随机写入转变为顺序写入底层存储芯片，因此存储引擎写入模式的影响不太明显【19】。但是，较低的写入放大率和减少的碎片对SSD仍然有利：更紧凑地表示数据可在可用的I/O带宽内提供更多的读取和写入请求。

LSM树的缺点

日志结构存储的缺点是压缩过程有时会干扰正在进行的读写操作。尽管存储引擎尝试逐步执行压缩而不影响并发访问，但是磁盘资源有限，所以很容易发生请求需要等待而磁盘完成昂贵的压缩操作。对吞吐量和平均响应时间的影响通常很小，但是在更高百分比的情况下（参阅“描述性能”），对日志结构化存储引擎的查询响应时间有时会相当长，而B树的行为则相对更具可预测性【28】。

压缩的另一个问题出现在高写入吞吐量：磁盘的有限写入带宽需要在初始写入（记录和刷新内存表到磁盘）和在后台运行的压缩线程之间共享。写入空数据库时，可以使用全磁盘带宽进行初始写入，但数据库越大，压缩所需的磁盘带宽就越多。

如果写入吞吐量很高，并且压缩没有仔细配置，压缩跟不上写入速率。在这种情况下，磁盘上未合并段的数量不断增加，直到磁盘空间用完，读取速度也会减慢，因为它们需要检查更多段文件。通常情况下，即使压缩无法跟上，基于SSTable的存储引擎也不会限制传入写入的速率，所以您需要进行明确的监控来检测这种情况【29,30】。

B树的一个优点是每个键只存在于索引中的一个位置，而日志结构化的存储引擎可能在不同的段中有相同键的多个副本。这个方面使得B树在想要提供强大的事务语义的数据库中很有吸引力：在许多关系数据库中，事务隔离是通过在键范围上使用锁来实现的，在B树索引中，这些锁可以直接连接到树【5】。在第7章中，我们将更详细地讨论这一点。

B树在数据库体系结构中是非常根深蒂固的，为许多工作负载提供始终如一的良好性能，所以它们不可能很快就会消失。在新的数据存储中，日志结构化索引变得越来越流行。没有快速和容易的规则来确定哪种类型的存储引擎对你的场景更好，所以值得进行一些经验上的测试

其他索引结构

到目前为止，我们只讨论了关键值索引，它们就像关系模型中的主键（**primary key**）索引。主键唯一标识关系表中的一行，或文档数据库中的一个文档或图形数据库中的一个顶点。数据库中的其他记录可以通过其主键（或ID）引用该行/文档/顶点，并且索引用于解析这样的引用。

有二级索引也很常见。在关系数据库中，您可以使用 `CREATE INDEX` 命令在同一个表上创建多个二级索引，而且这些索引通常对于有效地执行联接而言至关重要。例如，在第2章中的图2-1中，很可能在 `user_id` 列上有一个二级索引，以便您可以在每个表中找到属于同一用户的所有行。

一个二级索引可以很容易地从一个键值索引构建。主要的不同是键不是唯一的。即可能有许多行（文档，顶点）具有相同的键。这可以通过两种方式来解决：或者通过使索引中的每个值，成为匹配行标识符的列表（如全文索引中的发布列表），或者通过向每个索引添加行标识符来使每个关键字唯一。无论哪种方式，B树和日志结构索引都可以用作辅助索引。

将值存储在索引中

索引中的关键字是查询搜索的内容，但是该值可以是以下两种情况之一：它可以是所讨论的实际行（文档，顶点），也可以是对存储在别处的行的引用。在后一种情况下，行被存储的地方被称为堆文件（**heap file**），并且存储的数据没有特定的顺序（它可以是仅附加的，或者可以跟踪被删除的行以便用新数据覆盖它们后来）。堆文件方法很常见，因为它避免了在存在多个二级索引时复制数据：每个索引只引用堆文件中的一个位置，实际的数据保存在一个地方。 在不更改键的情况下更新值时，堆文件方法可以非常高效：只要新值不大于旧值，就可以覆盖该记录。如果新值更大，情况会更复杂，因为它可能需要移到堆中有足够空间的新位置。在这种情况下，要么所有的索引都需要更新，以指向记录的新堆位置，或者在旧堆位置留下一个转发指针【5】。

在某些情况下，从索引到堆文件的额外跳跃对读取来说性能损失太大，因此可能希望将索引行直接存储在索引中。这被称为聚集索引。例如，在MySQL的InnoDB存储引擎中，表的主键总是一个聚簇索引，二级索引用主键（而不是堆文件中的位置）【31】。在SQL Server中，可以为每个表指定一个聚簇索引【32】。

在聚集索引（**clustered index**）（在索引中存储所有行数据）和非聚集索引（**nonclustered index**）（仅在索引中存储对数据的引用）之间的折衷被称为包含列的索引（**index with included columns**）或覆盖索引（**covering index**），其存储表的一部分在索引内【33】。这允许通过单独使用索引来回答一些查询（这种情况叫做：索引覆盖（**cover**）了查询）【32】。

与任何类型的数据重复一样，聚簇和覆盖索引可以加快读取速度，但是它们需要额外的存储空间，并且会增加写入开销。数据库还需要额外的努力来执行事务保证，因为应用程序不应该因为重复而导致不一致。

多列索引

至今讨论的索引只是将一个键映射到一个值。如果我们需要同时查询一个表中的多个列（或文档中的多个字段），这显然是不够的。

最常见的多列索引被称为连接索引（**concatenated index**），它通过将一列的值追加到另一列后面，简单地将多个字段组合成一个键（索引定义中指定了字段的连接顺序）。这就像一个老式的纸质电话簿，它提供了一个从（姓，名）到电话号码的索引。由于排序顺序，索引可以用来查找所有具有特定姓氏的人，或所有具有特定姓-名组合的人。然而，如果你想找到所有具有特定名字的人，这个索引是没有用的。

多维索引（**multi-dimensional index**）是一种查询多个列的更一般的方法，这对于地理空间数据尤为重要。例如，餐厅搜索网站可能有一个数据库，其中包含每个餐厅的经度和纬度。当用户在地图上查看餐馆时，网站需要搜索用户正在查看的矩形地图区域内的所有餐馆。这需要一个二维范围查询，如下所示：

```sql
SELECT * FROM restaurants WHERE latitude > 51.4946 AND latitude < 51.5079
                           AND longitude > -0.1162 AND longitude < -0.1004;
```

一个标准的B树或者LSM树索引不能够高效地响应这种查询：它可以返回一个纬度范围内的所有餐馆（但经度可能是任意值），或者返回在同一个经度范围内的所有餐馆（但纬度可能是北极和南极之间的任意地方），但不能同时满足。

一种选择是使用空间填充曲线将二维位置转换为单个数字，然后使用常规B树索引【34】。更普遍的是，使用特殊化的空间索引，例如R树。例如，PostGIS使用PostgreSQL的通用Gist工具【35】将地理空间索引实现为R树。这里我们没有足够的地方来描述R树，但是有大量的文献可供参考。

一个有趣的主意是，多维索引不仅可以用于地理位置。例如，在电子商务网站上可以使用维度（红色，绿色，蓝色）上的三维索引来搜索特定颜色范围内的产品，也可以在天气观测数据库中搜索二维（日期，温度）的指数，以便有效地搜索2013年的温度在25至30°C之间的所有观测资料。使用一维索引，你将不得不扫描2013年的所有记录（不管温度如何），然后通过温度进行过滤，反之亦然。 二维索引可以同时通过时间戳和温度来收窄数据集。这个技术被HyperDex使用【36】。

全文搜索和模糊索引

到目前为止所讨论的所有索引都假定您有确切的数据，并允许您查询键的确切值或具有排序顺序的键的值范围。他们不允许你做的是搜索类似的键，如拼写错误的单词。这种模糊的查询需要不同的技术。

例如，全文搜索引擎通常允许搜索一个单词以扩展为包括该单词的同义词，忽略单词的语法变体，并且搜索在相同文档中彼此靠近的单词的出现，并且支持各种其他功能取决于文本的语言分析。为了处理文档或查询中的拼写错误，Lucene能够在一定的编辑距离内搜索文本

（编辑距离1意味着添加，删除或替换了一个字母）【37】。

正如“在SSTables中创建LSM树”中所提到的，Lucene为其词典使用了一个类似于SSTable的结构。这个结构需要一个小的内存索引，告诉查询在排序文件中哪个偏移量需要查找关键字。在LevelDB中，这个内存中的索引是一些键的稀疏集合，但在Lucene中，内存中的索引是键中字符的有限状态自动机，类似于trie 【38】。这个自动机可以转换成Levenshtein自动机，它支持在给定的编辑距离内有效地搜索单词【39】。

其他的模糊搜索技术正朝着文档分类和机器学习的方向发展。有关更多详细信息，请参阅信息检索教科书，例如【40】。

在内存中存储一切

本章到目前为止讨论的数据结构都是对磁盘限制的回答。与主内存相比，磁盘处理起来很尴尬。对于磁盘和SSD，如果要在读取和写入时获得良好性能，则需要仔细地布置磁盘上的数据。但是，我们容忍这种尴尬，因为磁盘有两个显着的优点：它们是耐用的（它们的内容在电源关闭时不会丢失），并且每GB的成本比RAM低。

随着RAM变得更便宜，每GB的成本价格被侵蚀了。许多数据集不是那么大，所以将它们全部保存在内存中是非常可行的，可能分布在多个机器上。这导致了内存数据库的发展。

某些内存中的键值存储（如Memcached）仅用于缓存，在重新启动计算机时丢失的数据是可以接受的。但其他内存数据库的目标是持久性，可以通过特殊的硬件（例如电池供电的RAM），将更改日志写入磁盘，将定时快照写入磁盘或通过复制内存来实现，记忆状态到其他机器。

内存数据库重新启动时，需要从磁盘或通过网络从副本重新加载其状态（除非使用特殊的硬件）。尽管写入磁盘，它仍然是一个内存数据库，因为磁盘仅用作耐久性附加日志，读取完全由内存提供。写入磁盘也具有操作优势：磁盘上的文件可以很容易地由外部实用程序进行备份，检查和分析。

诸如VoltDB，MemSQL和Oracle TimesTen等产品是具有关系模型的内存数据库，供应商声称，通过消除与管理磁盘上的数据结构相关的所有开销，他们可以提供巨大的性能改进【41,42】。 RAM Cloud是一个开源的内存键值存储器，具有持久性（对存储器中的数据以及磁盘上的数据使用日志结构化方法）【43】。 Redis和Couchbase通过异步写入磁盘提供了较弱的持久性。

反直觉的是，内存数据库的性能优势并不是因为它们不需要从磁盘读取的事实。即使是基于磁盘的存储引擎也可能永远不需要从磁盘读取，因为操作系统缓存最近在内存中使用了磁盘块。相反，它们更快的原因在于省去了将内存数据结构编码为磁盘数据结构的开销。【44】。

除了性能，内存数据库的另一个有趣的领域是提供难以用基于磁盘的索引实现的数据模型。例如，Redis为各种数据结构（如优先级队列和集合）提供了类似数据库的接口。因为它将所有数据保存在内存中，所以它的实现相对简单。

最近的研究表明，内存数据库体系结构可以扩展到支持比可用内存更大的数据集，而不必重新采用以磁盘为中心的体系结构【45】。所谓的反缓存（**anti-caching**）方法通过在内存不足的情况下将最近最少使用的数据从内存转移到磁盘，并在将来再次访问时将其重新加载到内存中。这与操作系统对虚拟内存和交换文件的操作类似，但数据库可以比操作系统更有效地管理内存，因为它可以按单个记录的粒度工作，而不是整个内存页面。尽管如此，这种方法仍然需要索引能完全放入内存中（就像本章开头的Bitcask例子）。

如果非易失性存储器（**NVM**）技术得到更广泛的应用，可能还需要进一步改变存储引擎设计【46】。目前这是一个新的研究领域，值得关注。

事务处理还是分析？

在业务数据处理的早期，对数据库的写入通常对应于正在进行的商业交易：进行销售，向供应商下订单，支付员工工资等等。随着数据库扩展到那些没有不涉及钱易手，术语交易仍然卡住，指的是形成一个逻辑单元的一组读写。 事务不一定具有ACID（原子性，一致性，隔离性和持久性）属性。事务处理只是意味着允许客户端进行低延迟读取和写入 —— 而不是批量处理作业，而这些作业只能定期运行（例如每天一次）。我们在第7章中讨论ACID属性，在第10章中讨论批处理。

即使数据库开始被用于许多不同类型的博客文章，游戏中的动作，地址簿中的联系人等等，基本访问模式仍然类似于处理业务事务。应用程序通常使用索引通过某个键查找少量记录。根据用户的输入插入或更新记录。由于这些应用程序是交互式的，因此访问模式被称为在线事务处理（**OLTP, OnLine Transaction Processing**）。

但是，数据库也开始越来越多地用于数据分析，这些数据分析具有非常不同的访问模式。通常，分析查询需要扫描大量记录，每个记录只读取几列，并计算汇总统计信息（如计数，总和或平均值），而不是将原始数据返回给用户。例如，如果您的数据是一个销售交易表，那么分析查询可能是：

- 一月份我们每个商店的总收入是多少？
- 我们在最近的推广活动中销售多少香蕉？
- 哪种品牌的婴儿食品最常与X品牌的尿布一起购买？

这些查询通常由业务分析师编写，并提供给帮助公司管理层做出更好决策（商业智能）的报告。为了区分这种使用数据库的事务处理模式，它被称为在线分析处理（**OLAP, OnLine Analytice Processing**）。【47】。OLTP和OLAP之间的区别并不总是清晰的，但是一些典型的特征在表3-1中列出。

表**3-1** 比较交易处理和分析系统的特点

属性	事务处理 OLTP	分析系统 OLAP
主要读取模式	查询少量记录，按键读取	在大批量记录上聚合
主要写入模式	随机访问，写入要求低延时	批量导入（ETL），事件流
主要用户	终端用户，通过Web应用	内部数据分析师，决策支持
处理的数据	数据的最新状态（当前时间点）	随时间推移的历史事件
数据集尺寸	GB ~ TB	TB ~ PB

起初，相同的数据库用于事务处理和分析查询。 SQL在这方面证明是非常灵活的：对于OLTP类型的查询以及OLAP类型的查询来说效果很好。尽管如此，在二十世纪八十年代末和九十年代初期，公司有停止使用OLTP系统进行分析的趋势，而是在单独的数据库上运行分析。这个单独的数据库被称为数据仓库（**data warehouse**）。

数据仓库

一个企业可能有几十个不同的交易处理系统：系统为面向客户的网站提供动力，控制实体商店的销售点（**checkout**）系统，跟踪仓库中的库存，规划车辆路线，管理供应商，管理员工等。这些系统中的每一个都是复杂的，需要一个人员去维护，所以系统最终都是自动运行的。

这些OLTP系统通常具有高度的可用性，并以低延迟处理事务，因为这些系统往往对业务运作至关重要。因此数据库管理员密切关注他们的OLTP数据库他们通常不愿意让业务分析人员在OLTP数据库上运行临时分析查询，因为这些查询通常很昂贵，扫描大部分数据集，这会损害同时执行的事务的性能。

相比之下，数据仓库是一个独立的数据库，分析人员可以查询他们心中的内容，而不影响OLTP操作【48】。数据仓库包含公司所有各种OLTP系统中的只读数据副本。从OLTP数据库中提取数据（使用定期的数据转储或连续的更新流），转换成适合分析的模式，清理并加载到数据仓库中。将数据存入仓库的过程称为“抽取-转换-加载（**ETL**）”，如图3-8所示。

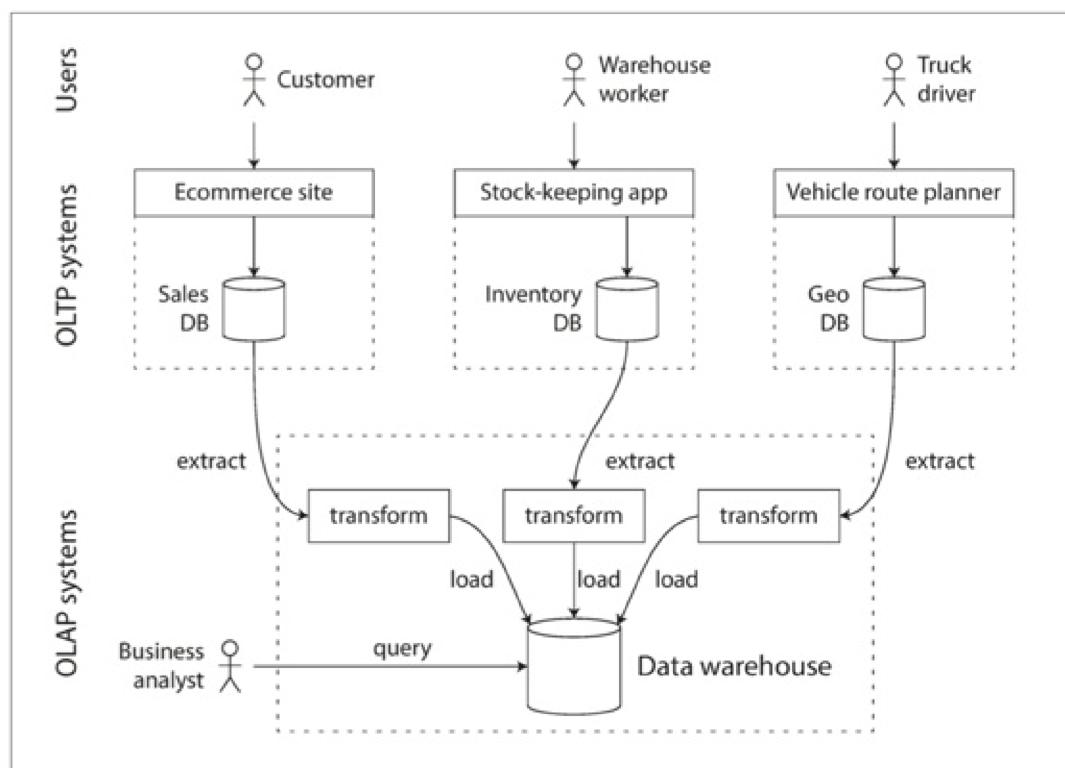


图**3-8 ETL**至数据仓库的简化提纲

几乎所有的大型企业都有数据仓库，但在小型企业中几乎闻所未闻。这可能是因为大多数小公司没有这么多不同的OLTP系统，大多数小公司只有少量的数据 —— 可以在传统的SQL数据库中查询，甚至可以在电子表格中分析。在一家大公司里，要做一些在一家小公司很简单的事情，需要很多繁重的工作。

使用单独的数据仓库，而不是直接查询OLTP系统进行分析的一大优势是数据仓库可针对分析访问模式进行优化。事实证明，本章前半部分讨论的索引算法对于OLTP来说工作得很好，但对于回答分析查询并不是很好。在本章的其余部分中，我们将看看为分析而优化的存储引擎。

OLTP数据库和数据仓库之间的分歧

数据仓库的数据模型通常是关系型的，因为SQL通常很适合分析查询。有许多图形数据分析工具可以生成SQL查询，可视化结果，并允许分析人员探索数据（通过下钻，切片和切块等操作）。

表面上，一个数据仓库和一个关系OLTP数据库看起来很相似，因为它们都有一个SQL查询接口。然而，系统的内部看起来可能完全不同，因为它们针对非常不同的查询模式进行了优化。现在许多数据库供应商都将重点放在支持事务处理或分析工作负载上，而不是两者都支持。

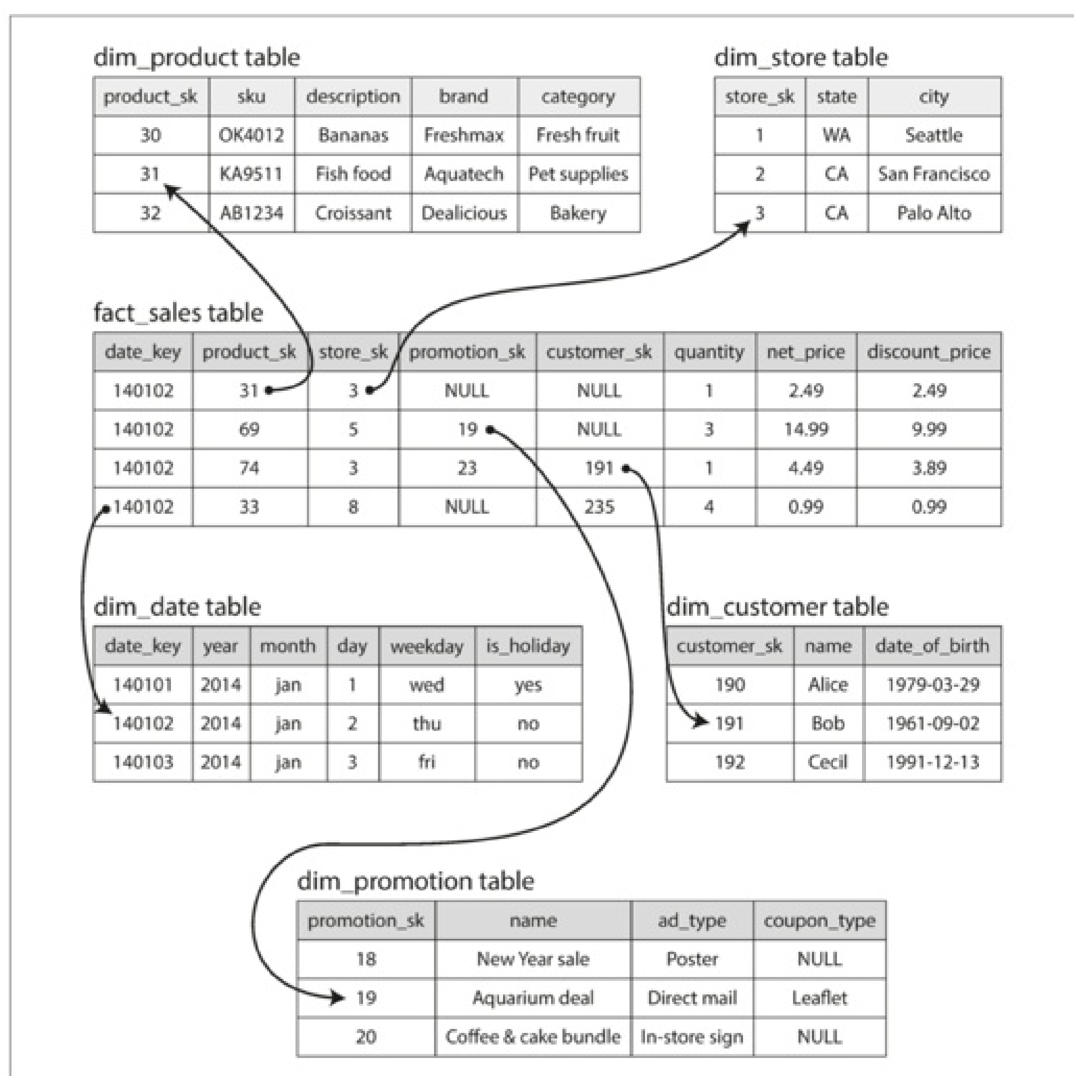
一些数据库（例如Microsoft SQL Server和SAP HANA）支持在同一产品中进行事务处理和数据仓库。但是，它们正在日益成为两个独立的存储和查询引擎，这些引擎正好可以通过一个通用的SQL接口访问【49,50,51】。

Teradata，Vertica，SAP HANA和ParAccel等数据仓库供应商通常使用昂贵的商业许可证销售他们的系统。 Amazon RedShift是ParAccel的托管版本。最近，大量的开源SQL-on-Hadoop项目已经出现，它们还很年轻，但是正在与商业数据仓库系统竞争。这些包括Apache Hive，Spark SQL，Cloudera Impala，Facebook Presto，Apache Tajo和Apache Drill【52,53】。其中一些是基于谷歌的Dremel [54]的想法。

星型和雪花型：分析的模式

正如第2章所探讨的，根据应用程序的需要，在事务处理领域中使用了大量不同的数据模型。另一方面，在分析中，数据模型的多样性则少得多。许多数据仓库都以相当公式化的方式使用，被称为星型模式（也称为维度建模【55】）。

图3-9中的示例模式显示了可能在食品零售商处找到的数据仓库。在模式的中心是一个所谓的事实表（在这个例子中，它被称为 `fact_sales` ）。事实表的每一行代表在特定时间发生的事件（这里，每一行代表客户购买的产品）。如果我们分析的是网站流量而不是零售量，则每行可能代表一个用户的页面浏览量或点击量。



dim_product table

product_sk	sku	description	brand	category
30	OK4012	Bananas	Freshmax	Fresh fruit
31	KA9511	Fish food	Aquatech	Pet supplies
32	AB1234	Croissant	Dealicious	Bakery

dim_store table

store_sk	state	city
1	WA	Seattle
2	CA	San Francisco
3	CA	Palo Alto

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	31	3	NULL	NULL	1	2.49	2.49
140102	69	5	19	NULL	3	14.99	9.99
140102	74	3	23	191	1	4.49	3.89
140102	33	8	NULL	235	4	0.99	0.99

dim_date table

date_key	year	month	day	weekday	is_holiday
140101	2014	jan	1	wed	yes
140102	2014	jan	2	thu	no
140103	2014	jan	3	fri	no

dim_customer table

customer_sk	name	date_of_birth
190	Alice	1979-03-29
191	Bob	1961-09-02
192	Cecil	1991-12-13

dim_promotion table

promotion_sk	name	ad_type	coupon_type
18	New Year sale	Poster	NULL
19	Aquarium deal	Direct mail	Leaflet
20	Coffee & cake bundle	In-store sign	NULL

图**3-9** 用于数据仓库的星型模式的示例

通常情况下，事实被视为单独的事件，因为这样可以在以后分析中获得最大的灵活性。但是，这意味着事实表可以变得非常大。像苹果，沃尔玛或eBay这样的大企业在其数据仓库中可能有几十PB的交易历史，其中大部分实际上是表【56】。

事实表中的一些列是属性，例如产品销售的价格和从供应商那里购买的成本（允许计算利润余额）。事实表中的其他列是对其他表（称为维表）的外键引用。由于事实表中的每一行都表示一个事件，因此这些维度代表事件的发生地点，时间，方式和原因。

例如，在图3-9中，其中一个维度是已售出的产品。 `dim_product` 表中的每一行代表一种待售产品，包括库存单位（**SKU**），说明，品牌名称，类别，脂肪含量，包装尺寸等。 `fact_sales` 表中的每一行都使用外部表明在特定交易中销售了哪些产品。（为了简单起见，如果客户一次购买几种不同的产品，则它们在事实表中被表示为单独的行）。

即使日期和时间通常使用维度表来表示，因为这允许对日期（诸如公共假期）的附加信息进行编码，从而允许查询区分假期和非假期的销售。

“星型模式”这个名字来源于这样一个事实，即当表关系可视化时，事实表在中间，由维表包围；与这些表的连接就像星星的光芒。

这个模板的变体被称为雪花模式，其中尺寸被进一步分解为子尺寸。例如，品牌和产品类别可能有单独的表格，并且 `dim_product` 表格中的每一行都可以将品牌和类别作为外键引用，而不是将它们作为字符串存储在 `dim_product` 表格中。雪花模式比星形模式更规范化，但是星形模式通常是首选，因为分析师使用它更简单【55】。

在典型的数据仓库中，表格通常非常宽泛：事实表格通常有100列以上，有时甚至有数百列【51】。维度表也可以是非常宽的，因为它们包括可能与分析相关的所有元数据——例如， `dim_store` 表可以包括在每个商店提供哪些服务的细节，它是否具有店内面包房，方形镜头，商店第一次开幕的日期，最后一次改造的时间，离最近的高速公路的距离等等。

列存储

如果事实表中有万亿行和数PB的数据，那么高效地存储和查询它们就成为一个具有挑战性的问题。维度表通常要小得多（数百万行），所以在本节中我们将主要关注事实的存储。

尽管事实表通常超过100列，但典型的数据仓库查询一次只能访问4个或5个查询（“ `SELECT *` ”查询很少用于分析）【51】。以例3-1中的查询为例：它访问了大量的行（在2013日历年中每次都有人购买水果或糖果），但只需访问 `fact_sales` 表的三列： `date_key, product_sk, quantity` 。查询忽略所有其他列。

例**3-1** 分析人们是否更倾向于购买新鲜水果或糖果，这取决于一周中的哪一天

```sql
SELECT
  dim_date.weekday,
  dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date ON fact_sales.date_key = dim_date.date_key
  JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
  dim_date.year = 2013 AND
  dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

我们如何有效地执行这个查询？

在大多数OLTP数据库中，存储都是以面向行的方式进行布局的：表格的一行中的所有值都相邻存储。文档数据库是相似的：整个文档通常存储为一个连续的字节序列。你可以在图3-1的CSV例子中看到这个。

为了处理像例3-1这样的查询，您可能在 `fact_sales.date_key` ， `fact_sales.product_sk` 上有索引，它们告诉存储引擎在哪里查找特定日期或特定产品的所有销售情况。但是，面向行的存储引擎仍然需要将所有这些行（每个包含超过100个属性）从磁盘加载到内存中，解析它们，并过滤掉那些不符合要求的条件。这可能需要很长时间。

面向列的存储背后的想法很简单：不要将所有来自一行的值存储在一起，而是将来自每一列的所有值存储在一起。如果每个列存储在一个单独的文件中，查询只需要读取和解析查询中使用的那些列，这可以节省大量的工作。这个原理如图3-10所示。

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents: 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents: 69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents: 4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents: NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents: NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents: 1, 3, 1, 5, 1, 3, 1, 1
net_price file contents: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

图**3-10** 使用列存储关系型数据，而不是行

列存储在关系数据模型中是最容易理解的，但它同样适用于非关系数据。例如，Parquet【57】是一种列式存储格式，支持基于Google的Dremel 【54】的文档数据模型。

面向列的存储布局依赖于包含相同顺序行的每个列文件。 因此，如果您需要重新组装整行，您可以从每个单独的列文件中获取第23项，并将它们放在一起形成表的第23行。

列压缩

除了仅从磁盘加载查询所需的列以外，我们还可以通过压缩数据来进一步降低对磁盘吞吐量的需求。幸运的是，面向列的存储通常很适合压缩。

看看图3-10中每一列的值序列：它们通常看起来是相当重复的，这是压缩的好兆头。根据列中的数据，可以使用不同的压缩技术。在数据仓库中特别有效的一种技术是位图编码，如图3-11所示。

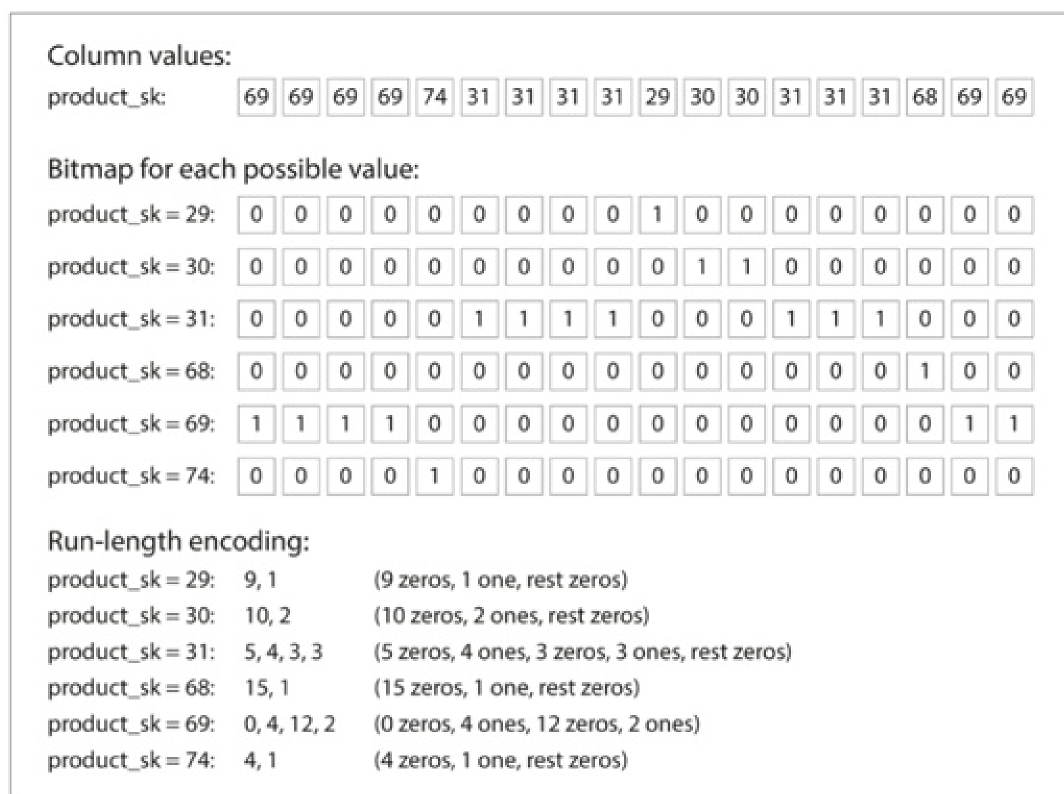


图**3-11** 压缩位图索引存储布局

通常情况下，一列中不同值的数量与行数相比较小（例如，零售商可能有数十亿的销售交易，但只有100,000个不同的产品）。现在我们可以得到一个有 n 个不同值的列，并把它转换成 n 个独立的位图：每个不同值的一个位图，每行一位。如果该行具有该值，则该位为 1 ，否则为 0 。

如果 n 非常小（例如，国家/地区列可能有大约200个不同的值），则这些位图可以每行存储一位。但是，如果n更大，大部分位图中将会有很多的零（我们说它们是稀疏的）。在这种情况下，位图可以另外进行游程编码，如图3-11底部所示。这可以使列的编码非常紧凑。

这些位图索引非常适合数据仓库中常见的各种查询。例如：

```
WHERE product_sk IN（30，68，69）
```

加载 `product_sk = 30` , `product_sk = 68` , `product_sk = 69` 的三个位图，并计算三个位图的按位或，这可以非常有效地完成。

```
WHERE product_sk = 31 AND store_sk = 3
```

加载 `product_sk = 31` 和 `store_sk = 3` 的位图，并逐位计算AND。 这是因为列按照相同的顺序包含行，因此一列的位图中的第 k 位对应于与另一列的位图中的第 k 位相同的行。

对于不同种类的数据，也有各种不同的压缩方案，但我们不会详细讨论它们，参见【58】的概述。

> 面向列的存储和列族
>
> Cassandra和HBase有一个列族的概念，他们从Bigtable继承【9】。然而，把它们称为面向列是非常具有误导性的：在每个列族中，它们将一行中的所有列与行键一起存储，并且不使用列压缩。因此，Bigtable模型仍然主要是面向行的。

内存带宽和向量处理

对于需要扫描数百万行的数据仓库查询来说，一个巨大的瓶颈是从磁盘获取数据到内存的带宽。但是，这不是唯一的瓶颈。分析数据库的开发人员也担心有效利用主存储器带宽到CPU缓存中的带宽，避免CPU指令处理流水线中的分支错误预测和泡沫，以及在现代中使用单指令多数据（SIMD）指令CPU 【59,60】。

除了减少需要从磁盘加载的数据量以外，面向列的存储布局也可以有效利用CPU周期。例如，查询引擎可以将大量压缩的列数据放在CPU的L1缓存中，然后在紧密的循环中循环（即没有函数调用）。一个CPU可以执行这样一个循环比代码要快得多，这个代码需要处理每个记录的大量函数调用和条件。列压缩允许列中的更多行适合相同数量的L1缓存。前面描述的按位“与”和“或”运算符可以被设计为直接在这样的压缩列数据块上操作。这种技术被称为矢量化处理【58,49】。

列存储中的排序顺序

在列存储中，存储行的顺序并不一定很重要。按插入顺序存储它们是最简单的，因为插入一个新行就意味着附加到每个列文件。但是，我们可以选择强制执行一个命令，就像我们之前对SSTables所做的那样，并将其用作索引机制。

注意，每列独自排序是没有意义的，因为那样我们就不会知道列中的哪些项属于同一行。我们只能重建一行，因为我们知道一列中的第k项与另一列中的第k项属于同一行。

相反，即使按列存储数据，也需要一次对整行进行排序。数据库的管理员可以使用他们对常见查询的知识来选择表格应该被排序的列。例如，如果查询通常以日期范围为目标，例如上个月，则可以将 `date_key` 作为第一个排序键。然后，查询优化器只能扫描上个月的行，这比扫描所有行要快得多。

第二列可以确定第一列中具有相同值的任何行的排序顺序。例如，如果 `date_key` 是图3-10中的第一个排序关键字，那么 `product_sk` 可能是第二个排序关键字，因此同一天的同一产品的所有销售都将在存储中组合在一起。这将有助于需要在特定日期范围内按产品对销售进行分组或过滤的查询。

排序顺序的另一个好处是它可以帮助压缩列。如果主要排序列没有多个不同的值，那么在排序之后，它将具有很长的序列，其中相同的值连续重复多次。一个简单的运行长度编码（就像我们用于图3-11中的位图一样）可以将该列压缩到几千字节 —— 即使表中有数十亿行。

第一个排序键的压缩效果最强。第二和第三个排序键会更混乱，因此不会有这么长时间的重复值。排序优先级下面的列以基本上随机的顺序出现，所以它们可能不会被压缩。但前几列排序仍然是一个整体。

几个不同的排序顺序

这个想法的巧妙扩展在C-Store中引入，并在商业数据仓库Vertica【61,62】中被采用。不同的查询受益于不同的排序顺序，为什么不以相同的方式存储相同的数据呢？无论如何，数据需要复制到多台机器，这样，如果一台机器发生故障，您不会丢失数据。您可能还需要存储以不同方式排序的冗余数据，以便在处理查询时，可以使用最适合查询模式的版本。

在一个面向列的存储中有多个排序顺序有点类似于在一个面向行的存储中有多个二级索引。但最大的区别在于面向行的存储将每一行保存在一个地方（在堆文件或聚簇索引中），二级索引只包含指向匹配行的指针。在列存储中，通常在其他地方没有任何指向数据的指针，只有包含值的列。

写入列存储

这些优化在数据仓库中是有意义的，因为大多数负载由分析人员运行的大型只读查询组成。面向列的存储，压缩和排序都有助于更快地读取这些查询。然而，他们有写更加困难的缺点。

使用B树的更新就地方法对于压缩的列是不可能的。如果你想在排序表的中间插入一行，你很可能不得不重写所有的列文件。由于行由列中的位置标识，因此插入必须始终更新所有列。

幸运的是，本章前面已经看到了一个很好的解决方案：LSM树。所有的写操作首先进入一个内存中的存储，在这里它们被添加到一个已排序的结构中，并准备写入磁盘。内存中的存储是面向行还是列的，这并不重要。当已经积累了足够的写入数据时，它们将与磁盘上的列文件合并，并批量写入新文件。这基本上是Vertica所做的【62】。

查询需要检查磁盘上的列数据和最近在内存中的写入，并将两者结合起来。但是，查询优化器隐藏了用户的这个区别。从分析师的角度来看，通过插入，更新或删除操作进行修改的数据会立即反映在后续查询中。

聚合：数据立方体和物化视图

并不是每个数据仓库都必定是一个列存储：传统的面向行的数据库和其他一些架构也被使用。然而，对于专门的分析查询，列式存储可以显着加快，所以它正在迅速普及【51,63】。

数据仓库的另一个值得一提的是物化汇总。如前所述，数据仓库查询通常涉及一个聚合函数，如SQL中的COUNT，SUM，AVG，MIN或MAX。如果相同的聚合被许多不同的查询使用，那么每次都可以通过原始数据来处理。为什么不缓存一些查询使用最频繁的计数或总和？

创建这种缓存的一种方式是物化视图。在关系数据模型中，它通常被定义为一个标准（虚拟）视图：一个类似于表的对象，其内容是一些查询的结果。不同的是，物化视图是查询结果的实际副本，写入磁盘，而虚拟视图只是写入查询的捷径。从虚拟视图读取时，SQL引擎会将其展开到视图的底层查询中，然后处理展开的查询。

当底层数据发生变化时，物化视图需要更新，因为它是数据的非规范化副本。数据库可以自动完成，但是这样的更新使得写入成本更高，这就是在OLTP数据库中不经常使用物化视图的原因。在读取繁重的数据仓库中，它们可能更有意义（不管它们是否实际上改善了读取性能取决于个别情况）。

物化视图的常见特例称为数据立方体或OLAP立方【64】。它是按不同维度分组的聚合网格。图3-12显示了一个例子。

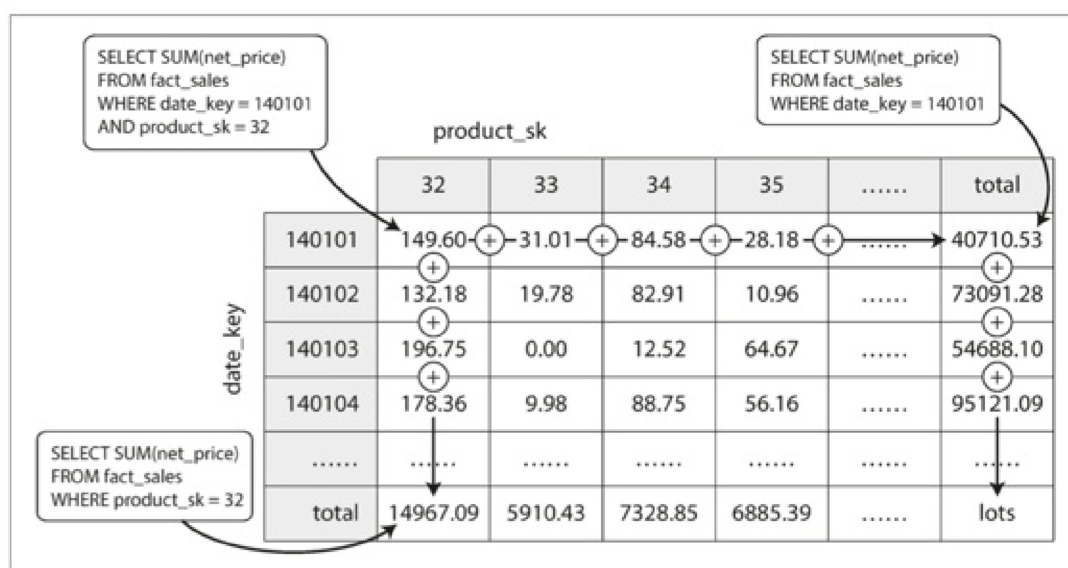


图**3-12** 数据立方的两个维度，通过求和聚合

想象一下，现在每个事实都只有两个维度表的外键——在图3-12中，这些是日期和产品。您现在可以绘制一个二维表格，一个轴线上的日期和另一个轴上的产品。每个单元包含具有该日期 - 产品组合的所有事实的属性（例如， `net_price` ）的聚集（例如， `SUM` ）。然后，您可以沿着每行或每列应用相同的汇总，并获得一个维度减少的汇总（按产品的销售额，无论日期，还是按日期销售，无论产品如何）。

一般来说，事实往往有两个以上的维度。在图3-9中有五个维度：日期，产品，商店，促销和客户。要想象一个五维超立方体是什么样子是很困难的，但是原理是一样的：每个单元格都包含特定日期（产品-商店-促销-客户）组合的销售。这些值可以在每个维度上重复概括。

物化数据立方体的优点是某些查询变得非常快，因为它们已经被有效地预先计算了。例如，如果您想知道每个商店的总销售额，则只需查看合适维度的总计，无需扫描数百万行。

缺点是数据立方体不具有查询原始数据的灵活性。例如，没有办法计算哪个销售比例来自成本超过100美元的项目，因为价格不是其中的一个维度。因此，大多数数据仓库试图保留尽可能多的原始数据，并将聚合数据（如数据立方体）仅用作某些查询的性能提升。

本章小结

在本章中，我们试图深入了解数据库如何处理存储和检索。将数据存储在数据库中会发生什么，以及稍后再次查询数据时数据库会做什么？

在高层次上，我们看到存储引擎分为两大类：优化事务处理（**OLTP**）和优化分析（**OLAP**）的类别。这些用例的访问模式之间有很大的区别：

- OLTP系统通常面向用户，这意味着他们可能会看到大量的请求。为了处理负载，应用程序通常只触及每个查询中的少量记录。应用程序使用某种键来请求记录，存储引擎使用索引来查找所请求的键的数据。磁盘寻道时间往往是这里的瓶颈。
- 数据仓库和类似的分析系统不太知名，因为它们主要由业务分析人员使用，而不是由最终用户使用。它们处理比OLTP系统少得多的查询量，但是每个查询通常要求很高，需要在短时间内扫描数百万条记录。磁盘带宽（不是查找时间）往往是瓶颈，列式存储是这种工作负载越来越流行的解决方案。

在OLTP方面，我们看到了来自两大主流学派的存储引擎：

日志结构学派

只允许附加到文件和删除过时的文件，但不会更新已经写入的文件。 Bitcask，SSTables，LSM树，LevelDB，Cassandra，HBase，Lucene等都属于这个组。

就地更新学派

将磁盘视为一组可以覆盖的固定大小的页面。 B树是这种哲学的最大的例子，被用在所有主要的关系数据库中，还有许多非关系数据库。

日志结构的存储引擎是相对较新的发展。他们的主要想法是，他们系统地将随机访问写入顺序写入磁盘，由于硬盘驱动器和固态硬盘的性能特点，可以实现更高的写入吞吐量。在完成OLTP方面，我们通过一些更复杂的索引结构和为保留所有数据而优化的数据库做了一个简短的介绍。

然后，我们从存储引擎的内部绕开，看看典型数据仓库的高级架构。这一背景说明了为什么分析工作负载与OLTP差别很大：当您的查询需要在大量行中顺序扫描时，索引的相关性就会降低很多。相反，非常紧凑地编码数据变得非常重要，以最大限度地减少查询需要从磁盘读取的数据量。我们讨论了列式存储如何帮助实现这一目标。

作为一名应用程序开发人员，如果您掌握了有关存储引擎内部的知识，那么您就能更好地了解哪种工具最适合您的特定应用程序。如果您需要调整数据库的调整参数，这种理解可以让您设想一个更高或更低的值可能会产生什么效果。

尽管本章不能让你成为一个特定存储引擎的调参专家，但它至少有大概率使你有了足够的概念与词汇储备去读懂数据库的文档，从而选择合适的数据库。

参考文献

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN: 978-0-201-00023-8

2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms*, 3rd edition. MIT Press, 2009. ISBN: 978-0-262-53305-8

3. Justin Sheehy and David Smith: “Bitcask: A Log-Structured Hash Table for Fast Key/Value Data,” Basho Technologies, April 2010.

4. Yinan Li, Bingsheng He, Robin Jun Yang, et al.: “Tree Indexing on Solid State Drives,” *Proceedings of the VLDB Endowment*, volume 3, number 1, pages 1195–1206, September 2010.

5. Goetz Graefe: “Modern B-Tree Techniques,” *Foundations and Trends in Databases*, volume 3, number 4, pages 203–402, August 2011. doi:10.1561/1900000028

6. Jeffrey Dean and Sanjay Ghemawat: “LevelDB Implementation Notes,” *leveldb.googlecode.com*.

7. Dhruba Borthakur: “The History of RocksDB,” *rocksdb.blogspot.com*, November 24, 2013.

8. Matteo Bertozzi: “Apache HBase I/O – HFile,” *blog.cloudera.com*, June, 29 2012.

9. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: "Bigtable: A Distributed Storage System for Structured Data," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.
10. Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil: "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica*, volume 33, number 4, pages 351–385, June 1996. doi:10.1007/s002360050048
11. Mendel Rosenblum and John K. Ousterhout: "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, volume 10, number 1, pages 26–52, February 1992. doi:10.1145/146941.146943
12. Adrien Grand: "What Is in a Lucene Index?," at *Lucene/Solr Revolution*, November 14, 2013.
13. Deepak Kandepet: "Hacking Lucene—The Index Format," *hackerlabs.org*, October 1, 2011.
14. Michael McCandless: "Visualizing Lucene's Segment Merges," *blog.mikemccandless.com*, February 11, 2011.
15. Burton H. Bloom: "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, volume 13, number 7, pages 422–426, July 1970. doi:10.1145/362686.362692
16. "Operating Cassandra: Compaction," Apache Cassandra Documentation v4.0, 2016.
17. Rudolf Bayer and Edward M. McCreight: "Organization and Maintenance of Large Ordered Indices," Boeing Scientific Research Laboratories, Mathematical and Information Sciences Laboratory, report no. 20, July 1970.
18. Douglas Comer: "The Ubiquitous B-Tree," *ACM Computing Surveys*, volume 11, number 2, pages 121–137, June 1979. doi:10.1145/356770.356776
19. Emmanuel Goossaert: "Coding for SSDs," *codecapsule.com*, February 12, 2014.
20. C. Mohan and Frank Levine: "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," at *ACM International Conference on Management of Data* (SIGMOD), June 1992. doi:10.1145/130283.130338
21. Howard Chu: "LDAP at Lightning Speed," at *Build Stuff '14*, November 2014.
22. Bradley C. Kuszmaul: "A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees," *tokutek.com*, April 22, 2014.

23. Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, et al.: "Designing Access Methods: The RUM Conjecture," at *19th International Conference on Extending Database Technology* (EDBT), March 2016. doi:10.5441/002/edbt.2016.42
24. Peter Zaitsev: "Innodb Double Write," *percona.com*, August 4, 2006.
25. Tomas Vondra: "On the Impact of Full-Page Writes," *blog.2ndquadrant.com*, November 23, 2016.
26. Mark Callaghan: "The Advantages of an LSM vs a B-Tree," *smalldatum.blogspot.co.uk*, January 19, 2016.
27. Mark Callaghan: "Choosing Between Efficiency and Performance with RocksDB," at *Code Mesh*, November 4, 2016.
28. Michi Mutsuzaki: "MySQL vs. LevelDB," *github.com*, August 2011.
29. Benjamin Coverston, Jonathan Ellis, et al.: "CASSANDRA-1608: Redesigned Compaction, *issues.apache.org*, July 2011.
30. Igor Canadi, Siying Dong, and Mark Callaghan: "RocksDB Tuning Guide," *github.com*, 2016.
31. *MySQL 5.7 Reference Manual*. Oracle, 2014.
32. *Books Online for SQL Server 2012*. Microsoft, 2012.
33. Joe Webb: "Using Covering Indexes to Improve Query Performance," *simple-talk.com*, 29 September 2008.
34. Frank Ramsak, Volker Markl, Robert Fenk, et al.: "Integrating the UB-Tree into a Database System Kernel," at *26th International Conference on Very Large Data Bases* (VLDB), September 2000.
35. The PostGIS Development Group: "PostGIS 2.1.2dev Manual," *postgis.net*, 2014.
36. Robert Escriva, Bernard Wong, and Emin Gün Sirer: "HyperDex: A Distributed, Searchable Key-Value Store," at *ACM SIGCOMM Conference*, August 2012. doi:10.1145/2377677.2377681
37. Michael McCandless: "Lucene's FuzzyQuery Is 100 Times Faster in 4.0," *blog.mikemccandless.com*, March 24, 2011.
38. Steffen Heinz, Justin Zobel, and Hugh E. Williams: "Burst Tries: A Fast, Efficient Data Structure for String Keys," *ACM Transactions on Information Systems*, volume 20, number 2, pages 192–223, April 2002. doi:10.1145/506309.506312

39. Klaus U. Schulz and Stoyan Mihov: "Fast String Correction with Levenshtein Automata," *International Journal on Document Analysis and Recognition*, volume 5, number 1, pages 67–85, November 2002. doi:10.1007/s10032-002-0082-8

40. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze: *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN: 978-0-521-86571-5, available online at *nlp.stanford.edu/IR-book*

41. Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: "The End of an Architectural Era (It's Time for a Complete Rewrite)," at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

42. "VoltDB Technical Overview White Paper," VoltDB, 2014.

43. Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout: "Log-Structured Memory for DRAM-Based Storage," at *12th USENIX Conference on File and Storage Technologies* (FAST), February 2014.

44. Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker: "OLTP Through the Looking Glass, and What We Found There," at *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376713

45. Justin DeBrabant, Andrew Pavlo, Stephen Tu, et al.: "Anti-Caching: A New Approach to Database Management System Architecture," *Proceedings of the VLDB Endowment*, volume 6, number 14, pages 1942–1953, September 2013.

46. Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor: "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2749441

47. Edgar F. Codd, S. B. Codd, and C. T. Salley: "Providing OLAP to User-Analysts: An IT Mandate," E. F. Codd Associates, 1993.

48. Surajit Chaudhuri and Umeshwar Dayal: "An Overview of Data Warehousing and OLAP Technology," *ACM SIGMOD Record*, volume 26, number 1, pages 65–74, March 1997. doi:10.1145/248603.248616

49. Per-Åke Larson, Cipri Clinciu, Campbell Fraser, et al.: "Enhancements to SQL Server Column Stores," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

50. Franz Färber, Norman May, Wolfgang Lehner, et al.: “The SAP HANA Database – An Architecture Overview,” *IEEE Data Engineering Bulletin*, volume 35, number 1, pages 28–33, March 2012.

51. Michael Stonebraker: “The Traditional RDBMS Wisdom Is (Almost Certainly) All Wrong,” presentation at *EPFL*, May 2013.

52. Daniel J. Abadi: “Classifying the SQL-on-Hadoop Solutions,” *hadapt.com*, October 2, 2013.

53. Marcel Kornacker, Alexander Behm, Victor Bittorf, et al.: “Impala: A Modern, Open-Source SQL Engine for Hadoop,” at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

54. Sergey Melnik, Andrey Gubarev, Jing Jing Long, et al.: “Dremel: Interactive Analysis of Web-Scale Datasets,” at *36th International Conference on Very Large Data Bases* (VLDB), pages 330–339, September 2010.

55. Ralph Kimball and Margy Ross: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, July 2013. ISBN: 978-1-118-53080-1

56. Derrick Harris: “Why Apple, eBay, and Walmart Have Some of the Biggest Data Warehouses You’ve Ever Seen,” *gigaom.com*, March 27, 2013.

57. Julien Le Dem: “Dremel Made Simple with Parquet,” *blog.twitter.com*, September 11, 2013.

58. Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos, et al.: “The Design and Implementation of Modern Column-Oriented Database Systems,” *Foundations and Trends in Databases*, volume 5, number 3, pages 197–280, December 2013. doi:10.1561/1900000024

59. Peter Boncz, Marcin Zukowski, and Niels Nes: “MonetDB/X100: Hyper-Pipelining Query Execution,” at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

60. Jingren Zhou and Kenneth A. Ross: “Implementing Database Operations Using SIMD Instructions,” at *ACM International Conference on Management of Data* (SIGMOD), pages 145–156, June 2002. doi:10.1145/564691.564709

61. Michael Stonebraker, Daniel J. Abadi, Adam Batkin, et al.: “C-Store: A Column-oriented DBMS,” at *31st International Conference on Very Large Data Bases* (VLDB), pages 553–564, September 2005.

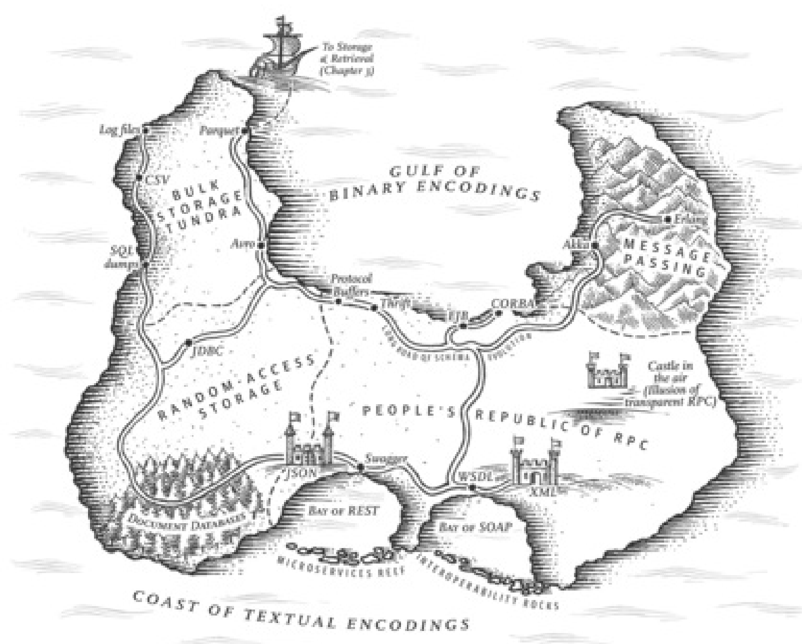
62. Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, et al.: “The Vertica Analytic Database: C-Store 7 Years Later,” *Proceedings of the VLDB Endowment*, volume 5, number 12, pages 1790–1801, August 2012.

63. Julien Le Dem and Nong Li: “Efficient Data Storage for Analytics with Apache Parquet 2.0,” at *Hadoop Summit*, San Jose, June 2014.

64. Jim Gray, Surajit Chaudhuri, Adam Bosworth, et al.: “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals,” *Data Mining and Knowledge Discovery*, volume 1, number 1, pages 29–53, March 2007. doi:10.1023/A:1009726021843

上一章	目录	下一章
第二章：数据模型与查询语言	设计数据密集型应用	第四章：编码与演化

4. 编码与演化



> 唯变所适
>
> ——以弗所的赫拉克利特，为柏拉图所引（公元前360年）

[TOC]

应用程序不可避免地随时间而变化。新产品的推出，对需求的深入理解，或者商业环境的变化，总会伴随着功能（**feature**）的增增改改。第一章介绍了可演化性(**evolvability**)的概念：应该尽力构建能灵活适应变化的系统（参阅“可演化性：拥抱变化”）。

在大多数情况下，修改应用程序的功能也意味着需要更改其存储的数据：可能需要使用新的字段或记录类型，或者以新方式展示现有数据。

我们在第二章讨论的数据模型有不同的方法来应对这种变化。关系数据库通常假定数据库中的所有数据都遵循一个模式：尽管可以更改该模式（通过模式迁移，即 `ALTER` 语句），但是在任何时间点都有且仅有一个正确的模式。相比之下，读时模式（**schema-on-read**）（或无模式（**schemaless**））数据库不会强制一个模式，因此数据库可以包含在不同时间写入的新老数据格式的混合（参阅 “文档模型中的模式灵活性” ）。

当数据格式（**format**）或模式（**schema**）发生变化时，通常需要对应用程序代码进行相应的更改（例如，为记录添加新字段，然后修改程序开始读写该字段）。但在大型应用程序中，代码变更通常不会立即完成：

- 对于服务端（**server-side**）应用程序，可能需要执行滚动升级 （**rolling upgrade**）（也称为阶段发布（**staged rollout**）），一次将新版本部署到少数几个节点，检查新版本是否运行正常，然后逐渐部完所有的节点。这样无需中断服务即可部署新版本，为频繁发布提供了可行性，从而带来更好的可演化性。
- 对于客户端（**client-side**）应用程序，升不升级就要看用户的心情了。用户可能相当长一段时间里都不会去升级软件。

这意味着，新旧版本的代码，以及新旧数据格式可能会在系统中同时共处。系统想要继续顺利运行，就需要保持双向兼容性：

向后兼容 ***(backward compatibility)***

新代码可以读旧数据。

向前兼容 ***(forward compatibility)***

旧代码可以读新数据。

向后兼容性通常并不难实现：新代码的作者当然知道由旧代码使用的数据格式，因此可以显示地处理它（最简单的办法是，保留旧代码即可读取旧数据）。

向前兼容性可能会更棘手，因为旧版的程序需要忽略新版数据格式中新增的部分。

本章中将介绍几种编码数据的格式，包括 JSON，XML，Protocol Buffers，Thrift和Avro。尤其将关注这些格式如何应对模式变化，以及它们如何对新旧代码数据需要共存的系统提供支持。然后将讨论如何使用这些格式进行数据存储和通信：在Web服务中，具象状态传输（**REST**）和远程过程调用（**RPC**），以及消息传递系统（如Actor和消息队列）。

编码数据的格式

程序通常（至少）使用两种形式的数据：

1. 在内存中，数据保存在对象，结构体，列表，数组，哈希表，树等中。 这些数据结构针对CPU的高效访问和操作进行了优化（通常使用指针）。
2. 如果要将数据写入文件，或通过网络发送，则必须将其编码（**encode**）为某种自包含的字节序列（例如，JSON文档）。 由于每个进程都有自己独立的地址空间，一个进程中的指针对任何其他进程都没有意义，所以这个字节序列表示会与通常在内存中使用的数据结构完全不同[i]。

> i. 除一些特殊情况外，例如某些内存映射文件或直接在压缩数据上操作（如“列压缩”中所述）。↩

所以，需要在两种表示之间进行某种类型的翻译。 从内存中表示到字节序列的转换称为编码（**Encoding**）（也称为序列化（**serialization**）或编组（**marshalling**）），反过来称为解码（**Decoding**）[ii]（解析（**Parsing**），反序列化（**deserialization**），反编组() **unmarshalling**）[译i]。

> ii. 请注意，编码（**encode**）与加密（**encryption**）无关。 本书不讨论加密。 ↩

> 译i. Marshal与Serialization的区别：Marshal不仅传输对象的状态，而且会一起传输对象的方法（相关代码）。 ↩

> 术语冲突
>
> 不幸的是，在第七章：事务（**Transaction**）的上下文里，序列化（**Serialization**）这个术语也出现了，而且具有完全不同的含义。尽管序列化可能是更常见的术语，为了避免术语重载，本书中坚持使用编码（**Encoding**）表达此含义。

这是一个常见的问题，因而有许多库和编码格式可供选择。 首先让我们概览一下。

语言特定的格式

许多编程语言都内建了将内存对象编码为字节序列的支持。例如，Java 有 `java.io.Serializable` 【1】，Ruby有 `Marshal` 【2】，Python有 `pickle` 【3】等等。许多第三方库也存在，例如 `Kryo for Java` 【4】。

这些编码库非常方便，可以用很少的额外代码实现内存对象的保存与恢复。但是它们也有一些深层次的问题：

- 这类编码通常与特定的编程语言深度绑定，其他语言很难读取这种数据。如果以这类编码存储或传输数据，那你就和这门语言绑死在一起了。并且很难将系统与其他组织的系统（可能用的是不同的语言）进行集成。
- 为了恢复相同对象类型的数据，解码过程需要实例化任意类的能力，这通常是安全问题的一个来源【5】：如果攻击者可以让应用程序解码任意的字节序列，他们就能实例化任意的类，这会允许他们做可怕的事情，如远程执行任意代码【6,7】。
- 在这些库中，数据版本控制通常是事后才考虑的。因为它们旨在快速简便地对数据进行编码，所以往往忽略了前向后向兼容性带来的麻烦问题。
- 效率（编码或解码所花费的CPU时间，以及编码结构的大小）往往也是事后才考虑的。例如，Java的内置序列化由于其糟糕的性能和臃肿的编码而臭名昭着【8】。

因此，除非临时使用，采用语言内置编码通常是一个坏主意。

JSON，XML和二进制变体

谈到可以被许多编程语言编写和读取的标准化编码，JSON和XML是显眼的竞争者。它们广为人知，广受支持，也“广受憎恶”。 XML经常被批评为过于冗长和不必要的复杂【9】。 JSON倍受欢迎，主要由于它在Web浏览器中的内置支持（通过成为JavaScript的一个子集）以及相对于XML的简单性。 CSV是另一种流行的与语言无关的格式，尽管功能较弱。

JSON，XML和CSV是文本格式，因此具有人类可读性（尽管语法是一个热门辩题）。除了表面的语法问题之外，它们也有一些微妙的问题：

- 数字的编码多有歧义之处。XML和CSV不能区分数字和字符串（除非引用外部模式）。JSON虽然区分字符串和数字，但不区分整数和浮点数，而且不能指定精度。
- 当处理大量数据时，这个问题更严重了。例如，大于2^{53}的整数不能在IEEE 754双精度浮点数中精确表示，因此在使用浮点数（例如JavaScript）的语言进行分析时，这些数字会变得不准确。 Twitter上有一个大于2^{53}的数字的例子，它使用一个64位的数字来标识每条推文。 Twitter API返回的JSON包含了两种推特ID，一个JSON数字，另一个是十进制字符串，以此避免JavaScript程序无法正确解析数字的问题【10】。
- JSON和XML对Unicode字符串（即人类可读的文本）有很好的支持，但是它们不支持二进制数据（不带字符编码(character encoding)的字节序列）。二进制串是很实用的功能，所以人们通过使用Base64将二进制数据编码为文本来绕开这个限制。模式然后用于表示该值应该被解释为Base64编码。这个工作，但它有点hacky，并增加了33％的数据大小。 XML 【11】和JSON 【12】都有可选的模式支持。这些模式语言相当强大，所以学习和实现起来相当复杂。 XML模式的使用相当普遍，但许多基于JSON的工具嫌麻烦才不会使用模式。由于数据的正确解释（例如数字和二进制字符串）取决于模式中的信息，因此不使用XML/JSON模式的应用程序可能需要对相应的编码/解码逻辑进行硬编码。
- CSV没有任何模式，因此应用程序需要定义每行和每列的含义。如果应用程序更改添加新的行或列，则必须手动处理该变更。 CSV也是一个相当模糊的格式（如果一个值包含逗号或换行符，会发生什么？）。尽管其转义规则已经被正式指定【13】，但并不是所有的解析器都正确的实现了标准。

尽管存在这些缺陷，但JSON，XML和CSV已经足够用于很多目的。特别是作为数据交换格式（即将数据从一个组织发送到另一个组织），它们很可能仍然很受欢迎。这种情况下，只要人们对格式是什么意见一致，格式多么美观或者高效就没有关系。让不同的组织达成一致的难度超过了其他大多数问题。

二进制编码

对于仅在组织内部使用的数据，使用最小公分母编码格式的压力较小。例如，可以选择更紧凑或更快的解析格式。虽然对小数据集来说，收益可以忽略不计，但一旦达到TB级别，数据格式的选择就会产生巨大的影响。

JSON比XML简洁，但与二进制格式一比，还是太占地方。这一事实导致大量二进制编码版本JSON & XML的出现，JSON（MessagePack，BSON，BJSON，UBJSON，BISON和Smile等）（例如WBXML和Fast Infoset）。这些格式已经被各种各样的领域所采用，但是没有一个

像JSON和XML的文本版本那样被广泛采用。

这些格式中的一些扩展了一组数据类型（例如，区分整数和浮点数，或者增加对二进制字符串的支持），另一方面，它们没有盖面JSON / XML的数据模型。特别是由于它们没有规定模式，所以它们需要在编码数据中包含所有的对象字段名称。也就是说，在例4-1中的JSON文档的二进制编码中，需要在某处包含字符串`userName`，`favoriteNumber`和`interest`。

例**4-1** 本章中用于展示二进制编码的示例记录

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

我们来看一个MessagePack的例子，它是一个JSON的二进制编码。图4-1显示了如果使用MessagePack【14】对例4-1中的JSON文档进行编码，则得到的字节序列。前几个字节如下：

1. 第一个字节`0x83`表示接下来是**3**个字段（低四位= `0x03`）的对象 **object**（高四位= `0x80`）。（如果想知道如果一个对象有15个以上的字段会发生什么情况，字段的数量塞不进4个bit里，那么它会用另一个不同的类型标识符，字段的数量被编码两个或四个字节）。
2. 第二个字节`0xa8`表示接下来是**8**字节长的字符串（最高四位= 0x08）。
3. 接下来八个字节是ASCII字符串形式的字段名称`userName`。由于之前已经指明长度，不需要任何标记来标识字符串的结束位置（或者任何转义）。
4. 接下来的七个字节对前缀为`0xa6`的六个字母的字符串值`Martin`进行编码，依此类推。

二进制编码长度为66个字节，仅略小于文本JSON编码所取的81个字节（删除了空白）。所有的JSON的二进制编码在这方面是相似的。空间节省了一丁点（以及解析加速）是否能弥补可读性的损失，谁也说不准。

在下面的章节中，能达到比这好得多的结果，只用32个字节对相同的记录进行编码。

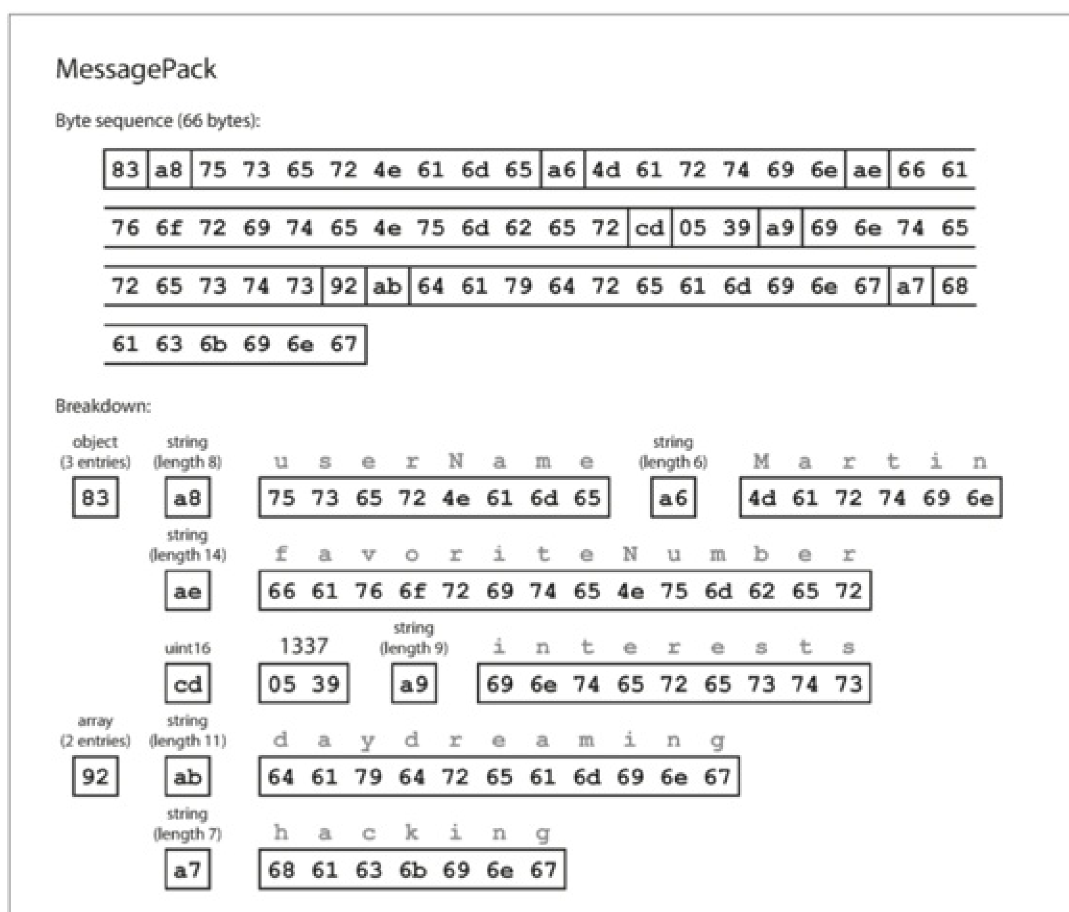


图**4-1** 使用**MessagePack**编码的记录（例**4-1**）

Thrift与Protocol Buffers

Apache Thrift【15】和Protocol Buffers（protobuf）【16】是基于相同原理的二进制编码库。Protocol Buffers最初是在Google开发的，Thrift最初是在Facebook开发的，并且在2007~2008年都是开源的【17】。Thrift和Protocol Buffers都需要一个模式来编码任何数据。要在Thrift的例4-1中对数据进行编码，可以使用Thrift接口定义语言（**IDL**）来描述模式，如下所示：

```
struct Person {
    1: required string       userName,
    2: optional i64          favoriteNumber,
    3: optional list<string> interests
}
```

Protocol Buffers的等效模式定义看起来非常相似：

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

Thrift和Protocol Buffers每一个都带有一个代码生成工具，它采用了类似于这里所示的模式定义，并且生成了以各种编程语言实现模式的类【18】。您的应用程序代码可以调用此生成的代码来对模式的记录进行编码或解码。 用这个模式编码的数据是什么样的？令人困惑的是，Thrift有两种不同的二进制编码格式[iii]，分别称为BinaryProtocol和CompactProtocol。先来看看BinaryProtocol。使用这种格式的编码来编码例4-1中的消息只需要59个字节，如图4-2所示【19】。

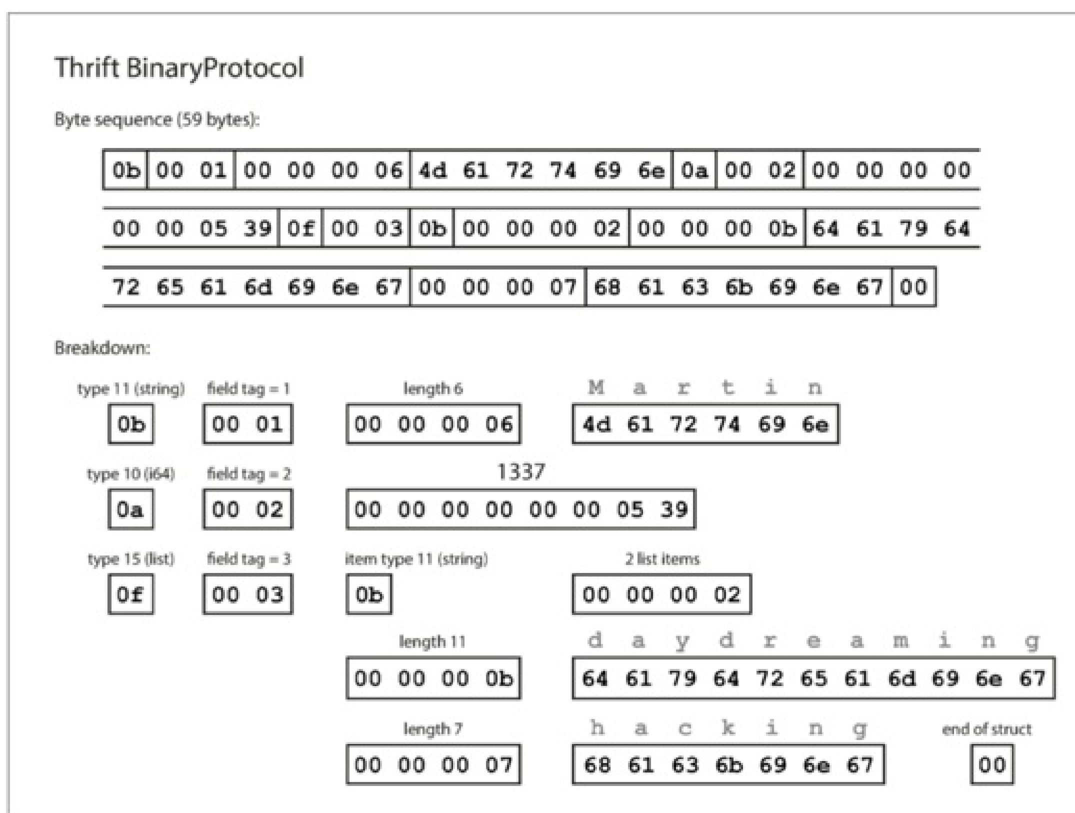


图**4-2** 使用**Thrift**二进制协议编码的记录

> iii. 实际上，Thrift有三种二进制协议：CompactProtocol和DenseProtocol，尽管DenseProtocol只支持C ++实现，所以不算作跨语言[18]。 除此之外，它还有两种不同的基于JSON的编码格式【19】。 真逗！↩

与图4-1类似，每个字段都有一个类型注释（用于指示它是一个字符串，整数，列表等），还可以根据需要指定长度（字符串的长度，列表中的项目数） 。出现在数据中的字符串 `(“Martin”, “daydreaming”, “hacking”)` 也被编码为ASCII（或者说，UTF-8），与之前类似。

与图4-1相比，最大的区别是没有字段名 `(userName, favoriteNumber, interest)` 。相反，编码数据包含字段标签，它们是数字 `(1, 2和3)` 。这些是模式定义中出现的数字。字段标记就像字段的别名 - 它们是说我们正在谈论的字段的一种紧凑的方式，而不必拼出字段名称。

Thrift CompactProtocol编码在语义上等同于BinaryProtocol，但是如图4-3所示，它只将相同的信息打包成只有34个字节。它通过将字段类型和标签号打包到单个字节中，并使用可变长度整数来实现。数字1337不是使用全部八个字节，而是用两个字节编码，每个字节的最高位

用来指示是否还有更多的字节来。这意味着-64到63之间的数字被编码为一个字节，-8192和8191之间的数字以两个字节编码，等等。较大的数字使用更多的字节。

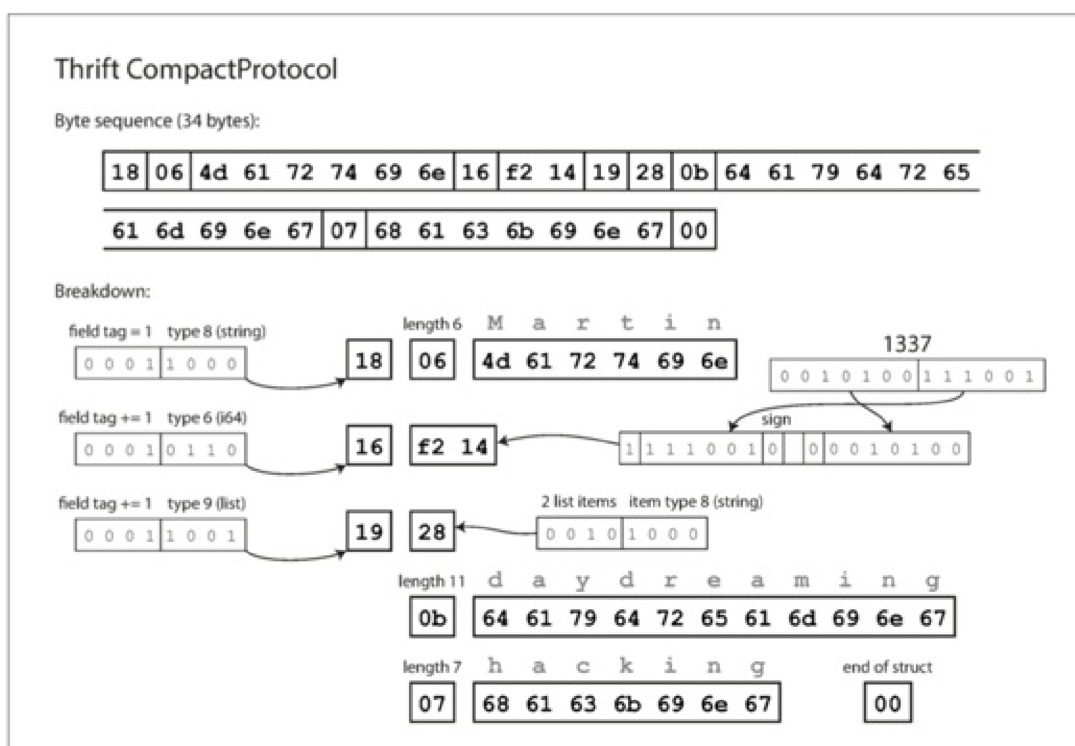


图4-3 使用Thrift压缩协议编码的记录

最后，Protocol Buffers（只有一种二进制编码格式）对相同的数据进行编码，如图4-4所示。它的打包方式稍有不同，但与Thrift的CompactProtocol非常相似。 Protobuf将同样的记录塞进了33个字节中。

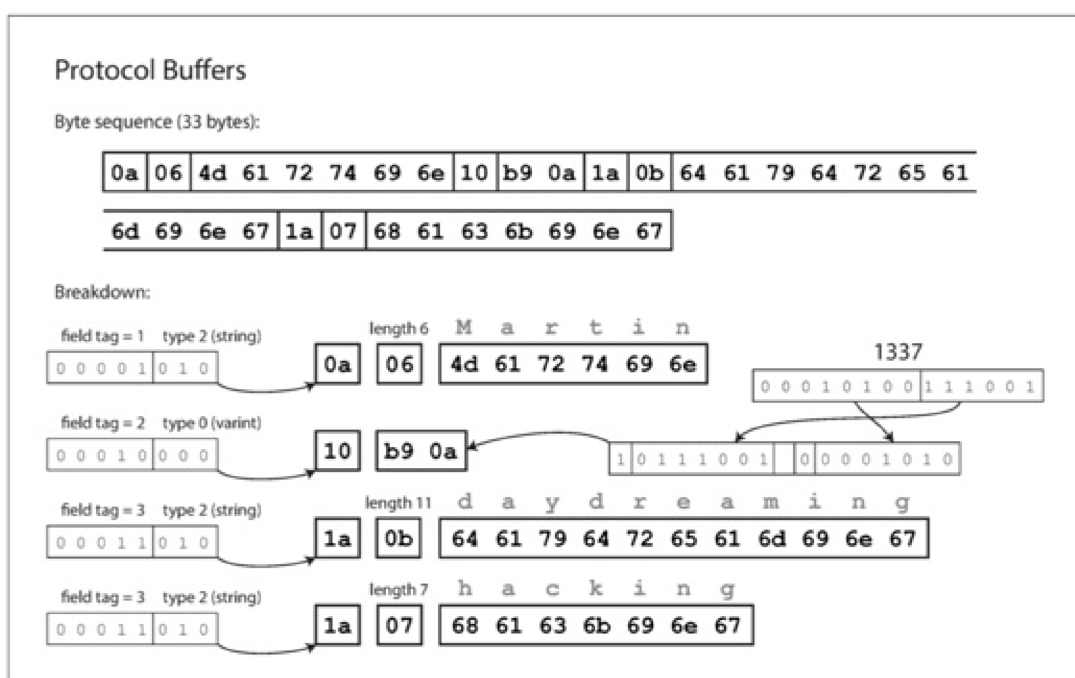


图4-4 使用Protobuf编码的记录

需要注意的一个细节：在前面所示的模式中，每个字段被标记为必需或可选，但是这对字段如何编码没有任何影响（二进制数据中没有任何字段指示是否需要字段）。所不同的是，如果未设置该字段，则所需的运行时检查将失败，这对于捕获错误非常有用。

字段标签和模式演变

我们之前说过，模式不可避免地需要随着时间而改变。我们称之为模式演变。 Thrift和Protocol Buffers如何处理模式更改，同时保持向后兼容性？

从示例中可以看出，编码的记录就是其编码字段的拼接。每个字段由其标签号码（样本模式中的数字1,2,3）标识，并用数据类型（例如字符串或整数）注释。如果没有设置字段值，则简单地从编码记录中省略。从中可以看到，字段标记对编码数据的含义至关重要。您可以更改架构中字段的名称，因为编码的数据永远不会引用字段名称，但不能更改字段的标记，因为这会使所有现有的编码数据无效。

您可以添加新的字段到架构，只要您给每个字段一个新的标签号码。如果旧的代码（不知道你添加的新的标签号码）试图读取新代码写入的数据，包括一个新的字段，其标签号码不能识别，它可以简单地忽略该字段。数据类型注释允许解析器确定需要跳过的字节数。这保持了前向兼容性：旧代码可以读取由新代码编写的记录。

向后兼容性呢？只要每个字段都有一个唯一的标签号码，新的代码总是可以读取旧的数据，因为标签号码仍然具有相同的含义。唯一的细节是，如果你添加一个新的领域，你不能要求。如果您要添加一个字段并将其设置为必需，那么如果新代码读取旧代码写入的数据，则该检查将失败，因为旧代码不会写入您添加的新字段。因此，为了保持向后兼容性，在模式的初始部署之后添加的每个字段必须是可选的或具有默认值。

删除一个字段就像添加一个字段，倒退和向前兼容性问题相反。这意味着您只能删除一个可选的字段（必填字段永远不能删除），而且您不能再次使用相同的标签号码（因为您可能仍然有数据写在包含旧标签号码的地方，而该字段必须被新代码忽略）。

数据类型和模式演变

如何改变字段的数据类型？这可能是可能的——检查文件的细节——但是有一个风险，值将失去精度或被扼杀。例如，假设你将一个32位的整数变成一个64位的整数。新代码可以轻松读取旧代码写入的数据，因为解析器可以用零填充任何缺失的位。但是，如果旧代码读取由新代码写入的数据，则旧代码仍使用32位变量来保存该值。如果解码的64位值不适合32位，则它将被截断。

Protobuf的一个奇怪的细节是，它没有列表或数组数据类型，而是有一个字段的重复标记（这是第三个选项旁边必要和可选）。如图4-4所示，重复字段的编码正如它所说的那样：同一个字段标记只是简单地出现在记录中。这具有很好的效果，可以将可选（单值）字段更改为重复（多值）字段。读取旧数据的新代码会看到一个包含零个或一个元素的列表（取决于该字段是否存在）。读取新数据的旧代码只能看到列表的最后一个元素。

Thrift有一个专用的列表数据类型，它使用列表元素的数据类型进行参数化。这不允许Protocol Buffers所做的从单值到多值的相同演变，但是它具有支持嵌套列表的优点。

Avro

Apache Avro 【20】是另一种二进制编码格式，与Protocol Buffers和Thrift有趣的不同。 它是作为Hadoop的一个子项目在2009年开始的，因为Thrift不适合Hadoop的用例【21】。

Avro也使用模式来指定正在编码的数据的结构。 它有两种模式语言：一种（Avro IDL）用于人工编辑，一种（基于JSON），更易于机器读取。

我们用Avro IDL编写的示例模式可能如下所示：

```
record Person {
    string                userName;
    union { null, long }  favoriteNumber = null;
    array<string>         interests;
}
```

等价的JSON表示：

```
{
    "type": "record",
    "name": "Person",
    "fields": [
        {"name": "userName", "type": "string"},
        {"name": "favoriteNumber", "type": ["null", "long"], "default": null},
        {"name": "interests", "type": {"type": "array", "items": "string"}
    ]
}
```

首先，请注意架构中没有标签号码。 如果我们使用这个模式编码我们的例子记录（例4-1），Avro二进制编码只有32个字节长，这是我们所见过的所有编码中最紧凑的。 编码字节序列的分解如图4-5所示。

如果您检查字节序列，您可以看到没有什么可以识别字段或其数据类型。 编码只是由连在一起的值组成。 一个字符串只是一个长度前缀，后跟UTF-8字节，但是在被包含的数据中没有任何内容告诉你它是一个字符串。 它可以是一个整数，也可以是其他的整数。 整数使用可变长度编码（与Thrift的CompactProtocol相同）进行编码。

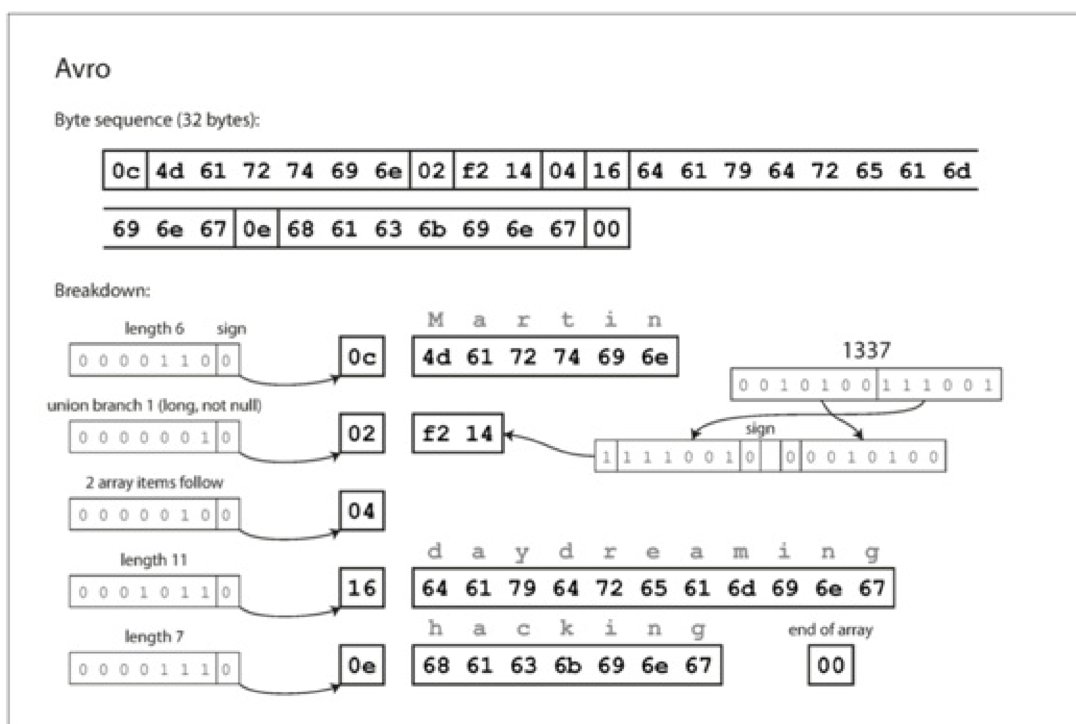


图**4-5** 使用**Avro**编码的记录

为了解析二进制数据，您按照它们出现在架构中的顺序遍历这些字段，并使用架构来告诉您每个字段的数据类型。这意味着如果读取数据的代码使用与写入数据的代码完全相同的模式，则只能正确解码二进制数据。阅读器和作者之间的模式不匹配意味着错误地解码数据。

那么，Avro如何支持模式演变呢？

作者模式与读者模式

有了Avro，当应用程序想要编码一些数据（将其写入文件或数据库，通过网络发送等）时，它使用它知道的任何版本的模式编码数据，例如，架构可能被编译到应用程序中。这被称为作者的模式。

当一个应用程序想要解码一些数据（从一个文件或数据库读取数据，从网络接收数据等）时，它希望数据在某个模式中，这就是读者的模式。这是应用程序代码所依赖的模式，在应用程序的构建过程中，代码可能是从该模式生成的。

Avro的关键思想是作者的模式和读者的模式不必是相同的 - 他们只需要兼容。当数据解码（读取）时，Avro库通过并排查看作者的模式和读者的模式并将数据从作者的模式转换到读者的模式来解决差异。 Avro规范【20】确切地定义了这种解析的工作原理，如图4-6所示。

例如，如果作者的模式和读者的模式的字段顺序不同，这是没有问题的，因为模式解析通过字段名匹配字段。如果读取数据的代码遇到出现在作者模式中但不在读者模式中的字段，则忽略它。如果读取数据的代码需要某个字段，但是作者的模式不包含该名称的字段，则使用在读者模式中声明的默认值填充。

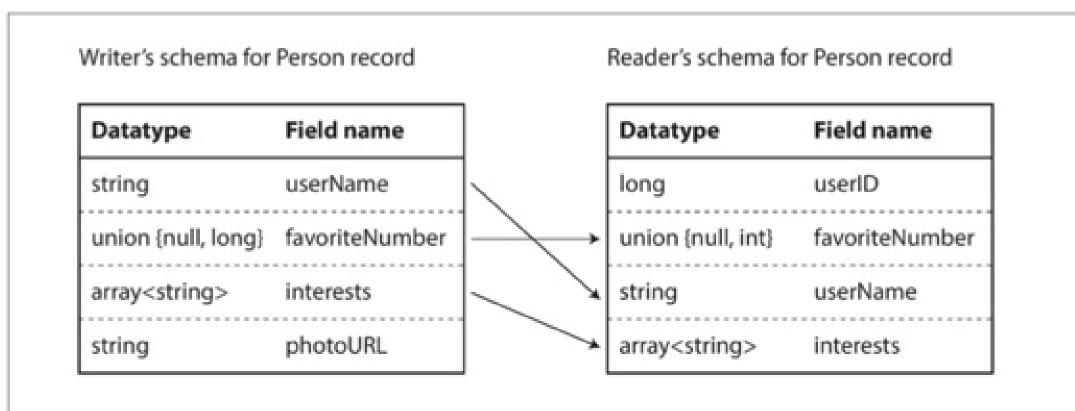


图**4-6** 一个**Avro Reader**解决读写模式的差异

模式演变规则

使用Avro，向前兼容性意味着您可以将新版本的架构作为编写器，并将旧版本的架构作为读者。相反，向后兼容意味着你可以有一个作为读者的新版本的模式和作为作者的旧版本。

为了保持兼容性，您只能添加或删除具有默认值的字段。（我们的Avro模式中的字段 `favourNumber` 的默认值为 `null` ）。例如，假设您添加一个默认值的字段，所以这个新的字段存在于新的模式中，而不是旧的。当使用新模式的阅读器读取使用旧模式写入的记录时，将为缺少的字段填充默认值。

如果你要添加一个没有默认值的字段，新的阅读器将无法读取旧作者写的数据，所以你会破坏向后兼容性。如果您要删除没有默认值的字段，旧的阅读器将无法读取新作者写入的数据，因此您会打破兼容性。在一些编程语言中，null是任何变量可以接受的默认值，但在Avro中并不是这样：如果要允许一个字段为 `null` ，则必须使用联合类型。例如， `union {null，long，string}` 字段;表示该字段可以是数字或字符串，也可以是 `null` 。如果它是union的分支之一，那么只能使用null作为默认值[iv]。这比默认情况下可以为 `null` 是更加冗长的，但是通过明确什么可以和不可以是什么，有助于防止错误的 `null` 【22】。

> iv. 确切地说，默认值必须是联合的第一个分支的类型，尽管这是Avro的特定限制，而不是联合类型的一般特征。↩

因此，Avro没有像Protocol Buffers和Thrift那样的 `optional` 和 `required` 标记（它有联合类型和默认值）。

只要Avro可以转换类型，就可以改变字段的数据类型。更改字段的名称是可能的，但有点棘手：读者的模式可以包含字段名称的别名，所以它可以匹配旧作家的模式字段名称与别名。这意味着更改字段名称是向后兼容的，但不能向前兼容。同样，向联合类型添加分支也是向后兼容的，但不能向前兼容。

但作者模式到底是什么？

到目前为止，我们已经讨论了一个重要的问题：读者如何知道作者的模式是哪一部分数据被编码的？我们不能只将整个模式包括在每个记录中，因为模式可能比编码的数据大得多，从而使二进制编码节省的所有空间都是徒劳的。 答案取决于Avro使用的上下文。举几个例子：

- 有很多记录的大文件

  Avro的一个常见用途 - 尤其是在Hadoop环境中 - 用于存储包含数百万条记录的大文件，所有记录都使用相同的模式进行编码。 （我们将在第10章讨论这种情况。）在这种情况下，该文件的作者可以在文件的开头只包含一次作者的模式。 Avro指定一个文件格式（对象容器文件）来做到这一点。

- 支持独立写入的记录的数据库

  在一个数据库中，不同的记录可能会在不同的时间点使用不同的作者的模式编写 - 你不能假定所有的记录都有相同的模式。最简单的解决方案是在每个编码记录的开始处包含一个版本号，并在数据库中保留一个模式版本列表。读者可以获取记录，提取版本号，然后从数据库中获取该版本号的作者模式。使用该作者的模式，它可以解码记录的其余部分。 （例如Espresso 【23】就是这样工作的。）

- 通过网络连接发送记录

  当两个进程通过双向网络连接进行通信时，他们可以在连接设置上协商模式版本，然后在连接的生命周期中使用该模式。 Avro RPC协议（参阅“通过服务的数据流：REST和RPC”）如此工作。

具有模式版本的数据库在任何情况下都是非常有用的，因为它充当文档并为您提供了检查模式兼容性的机会【24】。作为版本号，你可以使用一个简单的递增整数，或者你可以使用模式的散列。

动态生成的模式

与Protocol Buffers和Thrift相比，Avro方法的一个优点是架构不包含任何标签号码。但为什么这很重要？在模式中保留一些数字有什么问题？

不同之处在于Avro对动态生成的模式更友善。例如，假如你有一个关系数据库，你想要把它的内容转储到一个文件中，并且你想使用二进制格式来避免前面提到的文本格式（JSON，CSV，SQL）的问题。如果你使用Avro，你可以很容易地从关系模式生成一个Avro模式（在我们之前看到的JSON表示中），并使用该模式对数据库内容进行编码，并将其全部转储到Avro对象容器文件【25】中。您为每个数据库表生成一个记录模式，每个列成为该记录中的一个字段。数据库中的列名称映射到Avro中的字段名称。

现在，如果数据库模式发生变化（例如，一个表中添加了一列，删除了一列），则可以从更新的数据库模式生成新的Avro模式，并在新的Avro模式中导出数据。数据导出过程不需要注意模式的改变 - 每次运行时都可以简单地进行模式转换。任何读取新数据文件的人都会看到记

录的字段已经改变，但是由于字段是通过名字来标识的，所以更新的作者的模式仍然可以与旧的读者模式匹配。

相比之下，如果您为此使用Thrift或Protocol Buffers，则字段标记可能必须手动分配：每次数据库模式更改时，管理员都必须手动更新从数据库列名到字段标签。（这可能会自动化，但模式生成器必须非常小心，不要分配以前使用的字段标记。）这种动态生成的模式根本不是Thrift或Protocol Buffers的设计目标，而是为Avro。

代码生成和动态类型的语言

Thrift和Protobuf依赖于代码生成：在定义了模式之后，可以使用您选择的编程语言生成实现此模式的代码。这在Java，C ++或C＃等静态类型语言中很有用，因为它允许将高效的内存中结构用于解码的数据，并且在编写访问数据结构的程序时允许在IDE中进行类型检查和自动完成。

在动态类型编程语言（如JavaScript，Ruby或Python）中，生成代码没有太多意义，因为没有编译时类型检查器来满足。代码生成在这些语言中经常被忽视，因为它们避免了明确的编译步骤。而且，对于动态生成的模式（例如从数据库表生成的Avro模式），代码生成对获取数据是一个不必要的障碍。

Avro为静态类型编程语言提供了可选的代码生成功能，但是它也可以在不生成任何代码的情况下使用。如果你有一个对象容器文件（它嵌入了作者的模式），你可以简单地使用Avro库打开它，并以与查看JSON文件相同的方式查看数据。该文件是自描述的，因为它包含所有必要的元数据。

这个属性特别适用于动态类型的数据处理语言如Apache Pig 【26】。在Pig中，您可以打开一些Avro文件，开始分析它们，并编写派生数据集以Avro格式输出文件，而无需考虑模式。

模式的优点

正如我们所看到的，Protocol Buffers，Thrift和Avro都使用模式来描述二进制编码格式。他们的模式语言比XML模式或者JSON模式简单得多，它支持更详细的验证规则（例如，“该字段的字符串值必须与该正则表达式匹配”或“该字段的整数值必须在0和100之间“）。由于Protocol Buffers，Thrift和Avro实现起来更简单，使用起来也更简单，所以它们已经发展到支持相当广泛的编程语言。

这些编码所基于的想法绝不是新的。例如，它们与ASN.1有很多相似之处，它是1984年首次被标准化的模式定义语言【27】。它被用来定义各种网络协议，其二进制编码（DER）仍然被用于编码SSL证书（X.509），例如【28】。 ASN.1支持使用标签号码的模式演进，类似于Protocol Buf-fers和Thrift 【29】。然而，这也是非常复杂和严重的文件记录，所以ASN.1可能不是新应用程序的好选择。

许多数据系统也为其数据实现某种专有的二进制编码。例如，大多数关系数据库都有一个网络协议，您可以通过该协议向数据库发送查询并获取响应。这些协议通常特定于特定的数据库，并且数据库供应商提供将来自数据库的网络协议的响应解码为内存数据结构的驱动程序（例如使用ODBC或JDBC API）。

所以，我们可以看到，尽管JSON，XML和CSV等文本数据格式非常普遍，但基于模式的二进制编码也是一个可行的选择。他们有一些很好的属性：

- 它们可以比各种“二进制JSON”变体更紧凑，因为它们可以省略编码数据中的字段名称。
- 模式是一种有价值的文档形式，因为模式是解码所必需的，所以可以确定它是最新的（而手动维护的文档可能很容易偏离现实）。
- 保留模式数据库允许您在部署任何内容之前检查模式更改的向前和向后兼容性。
- 对于静态类型编程语言的用户来说，从模式生成代码的能力是有用的，因为它可以在编译时进行类型检查。

总而言之，模式进化允许与JSON数据库提供的无模式/模式读取相同的灵活性（请参阅第39页的“文档模型中的模式灵活性”），同时还可以更好地保证数据和更好的工具。

数据流的类型

在本章的开始部分，我们曾经说过，无论何时您想要将某些数据发送到不共享内存的另一个进程，例如，只要您想通过网络发送数据或将其写入文件，就需要将它编码为一个字节序列。然后我们讨论了做这个的各种不同的编码。 我们讨论了向前和向后的兼容性，这对于可演化性来说非常重要（通过允许您独立升级系统的不同部分，而不必一次改变所有内容，可以轻松地进行更改）。兼容性是编码数据的一个进程和解码它的另一个进程之间的一种关系。

这是一个相当抽象的概念 - 数据可以通过多种方式从一个流程流向另一个流程。谁编码数据，谁解码？在本章的其余部分中，我们将探讨数据如何在流程之间流动的一些最常见的方式：

- 通过数据库（参阅“通过数据库的数据流”）
- 通过服务调用（参阅“通过服务传输数据流：REST和RPC”）
- 通过异步消息传递（参阅“消息传递数据流”）

数据库中的数据流

在数据库中，写入数据库的过程对数据进行编码，从数据库读取的过程对数据进行解码。可能只有一个进程访问数据库，在这种情况下，读者只是相同进程的后续版本 - 在这种情况下，您可以考虑将数据库中的内容存储为向未来的自我发送消息。

向后兼容性显然是必要的。否则你未来的自己将无法解码你以前写的东西。

一般来说，几个不同的进程同时访问数据库是很常见的。这些进程可能是几个不同的应用程序或服务，或者它们可能只是几个相同服务的实例（为了可扩展性或容错性而并行运行）。无论哪种方式，在应用程序发生变化的环境中，访问数据库的某些进程可能会运行较新的代码，有些进程可能会运行较旧的代码，例如，因为新版本当前正在部署在滚动升级，所以有些实例已经更新，而其他实例尚未更新。

这意味着数据库中的一个值可能会被更新版本的代码写入，然后被仍旧运行的旧版本的代码读取。因此，数据库也经常需要向前兼容。

但是，还有一个额外的障碍。假设您将一个字段添加到记录模式，并且较新的代码将该新字段的值写入数据库。随后，旧版本的代码（尚不知道新字段）将读取记录，更新记录并将其写回。在这种情况下，理想的行为通常是旧代码保持新的领域完整，即使它不能被解释。

前面讨论的编码格式支持未知域的保存，但是有时候需要在应用程序层面保持谨慎，如图4-7所示。例如，如果将数据库值解码为应用程序中的模型对象，稍后重新编码这些模型对象，那么未知字段可能会在该翻译过程中丢失。

解决这个问题不是一个难题，你只需要意识到它。

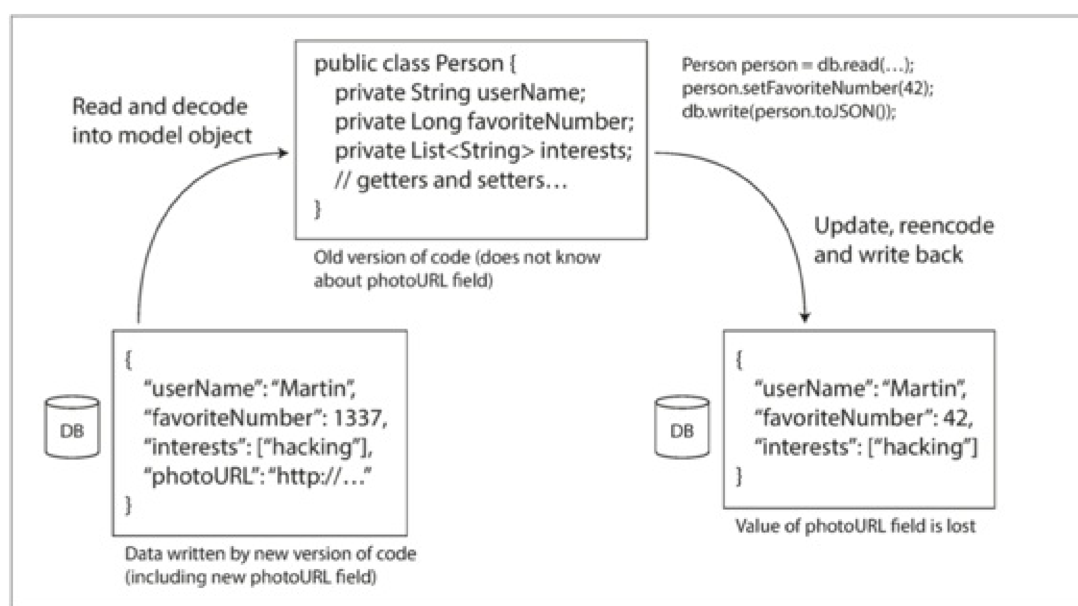


图4-7 当较旧版本的应用程序更新以前由较新版本的应用程序编写的数据时，如果不小心，数据可能会丢失。

在不同的时间写入不同的值

数据库通常允许任何时候更新任何值。这意味着在一个单一的数据库中，可能有一些价值是五毫秒前写的，而一些价值是五年前写的。

在部署应用程序的新版本（至少是服务器端应用程序）时，您可能会在几分钟内完全用新版本替换旧版本。数据库内容也是如此：五年前的数据仍然存在于原始编码中，除非您已经明确地重写了它。这种观察有时被总结为数据超出代码。

将数据重写（迁移）到一个新的模式当然是可能的，但是在一个大数据集上执行是一个昂贵的事情，所以大多数数据库如果可能的话就避免它。大多数关系数据库都允许简单的模式更改，例如添加一个默认值为空的新列，而不重写现有数据[v]读取旧行时，数据库将填充编码数据中缺少的任何列的空值在磁盘上。LinkedIn的文档数据库Espresso使用Avro存储，允许它使用Avro的模式演变规则【23】。

因此，架构演变允许整个数据库看起来好像是用单个模式编码的，即使底层存储可能包含用模式的各种历史版本编码的记录。

> v. 除了MySQL，即使并非真的必要，它也经常会重写整个表，正如“文档模型中的架构灵活性”中所提到的。↩

归档存储

也许您不时为数据库创建一个快照，例如备份或加载到数据仓库（参阅“数据仓库”）。在这种情况下，即使源数据库中的原始编码包含来自不同时代的模式版本的混合，数据转储通常也将使用最新模式进行编码。既然你正在复制数据，那么你可能会一直对数据的副本进行编码。

由于数据转储是一次写入的，而且以后是不可变的，所以Avro对象容器文件等格式非常适合。这也是一个很好的机会，可以将数据编码为面向分析的列式格式，例如Parquet（请参阅第97页的“列压缩”）。

在第10章中，我们将详细讨论在档案存储中使用数据。

服务中的数据流：REST 与 RPC

当您需要通过网络进行通信的进程时，安排该通信的方式有几种。最常见的安排是有两个角色：客户端和服务器。服务器通过网络公开API，并且客户端可以连接到服务器以向该API发出请求。服务器公开的API被称为服务。

Web以这种方式工作：客户（Web浏览器）向Web服务器发出请求，使GET请求下载HTML，CSS，JavaScript，图像等，并向POST请求提交数据到服务器。API包含一组标准的协议和数据格式（HTTP，URL，SSL/TLS，HTML等）。由于网络浏览器，网络服务器和网站作者大多同意这些标准，您可以使用任何网络浏览器访问任何网站（至少在理论上！）。

Web浏览器不是唯一的客户端类型。例如，在移动设备或桌面计算机上运行的本地应用程序也可以向服务器发出网络请求，并且在Web浏览器内运行的客户端JavaScript应用程序可以使用XMLHttpRequest成为HTTP客户端（该技术被称为Ajax【30】）。在这种情况下，服务器的响应通常不是用于显示给人的HTML，而是用于便于客户端应用程序代码（如JSON）进一步处理的编码数据。尽管HTTP可能被用作传输协议，但顶层实现的API是特定于应用程序的，客户端和服务器需要就该API的细节达成一致。

此外，服务器本身可以是另一个服务的客户端（例如，典型的Web应用服务器充当数据库的客户端）。这种方法通常用于将大型应用程序按照功能区域分解为较小的服务，这样当一个服务需要来自另一个服务的某些功能或数据时，就会向另一个服务发出请求。这种构建应用程序的方式传统上被称为面向服务的体系结构（**service-oriented architecture**，**SOA**），最近被改进和更名为微服务架构 【31,32】。

在某些方面，服务类似于数据库：它们通常允许客户端提交和查询数据。但是，虽然数据库允许使用我们在第2章 中讨论的查询语言进行任意查询，但是服务公开了一个特定于应用程序的API，它只允许由服务的业务逻辑（应用程序代码）预定的输入和输出【33】。这种限制提供了一定程度的封装：服务可以对客户可以做什么和不可以做什么施加细粒度的限制。

面向服务/微服务架构的一个关键设计目标是通过使服务独立部署和演化来使应用程序更易于更改和维护。例如，每个服务应该由一个团队拥有，并且该团队应该能够经常发布新版本的服务，而不必与其他团队协调。换句话说，我们应该期望服务器和客户端的旧版本和新版本同时运行，因此服务器和客户端使用的数据编码必须在不同版本的服务API之间兼容——正是我们所做的本章一直在谈论。

Web服务

当服务使用**HTTP**作为底层通信协议时，可称之为**Web**服务。这可能是一个小错误，因为Web服务不仅在Web上使用，而且在几个不同的环境中使用。例如：

1. 运行在用户设备上的客户端应用程序（例如，移动设备上的本地应用程序，或使用Ajax的JavaScript web应用程序）通过HTTP向服务发出请求。这些请求通常通过公共互联网进行。
2. 一种服务向同一组织拥有的另一项服务提出请求，这些服务通常位于同一数据中心内，作为面向服务/微型架构的一部分。 （支持这种用例的软件有时被称为中间件（**middleware**）。）
3. 一种服务通过互联网向不同组织所拥有的服务提出请求。这用于不同组织后端系统之间的数据交换。此类别包括由在线服务（如信用卡处理系统）提供的公共API，或用于共享访问用户数据的OAuth。

有两种流行的Web服务方法：REST和SOAP。他们在哲学方面几乎是截然相反的，往往是各自支持者之间的激烈辩论（即使在每个阵营内也有很多争论。 例如，**HATEOAS**（超媒体作为应用程序状态的引擎）经常引发讨论【35】。）

REST不是一个协议，而是一个基于HTTP原则的设计哲学【34,35】。它强调简单的数据格式，使用URL来标识资源，并使用HTTP功能进行缓存控制，身份验证和内容类型协商。与SOAP相比，REST已经越来越受欢迎，至少在跨组织服务集成的背景下【36】，并经常与微服务相关[31]。根据REST原则设计的API称为RESTful。

相比之下，SOAP是用于制作网络API请求的基于XML的协议（ 尽管首字母缩写词相似，SOAP并不是SOA的要求。 SOAP是一种特殊的技术，而SOA是构建系统的一般方法。）。虽然它最常用于HTTP，但其目的是独立于HTTP，并避免使用大多数HTTP功能。相反，它带

有庞大而复杂的多种相关标准（Web服务框架，称为 `WS-*` ），它们增加了各种功能【37】。

SOAP Web服务的API使用称为Web服务描述语言（WSDL）的基于XML的语言来描述。WSDL支持代码生成，客户端可以使用本地类和方法调用（编码为XML消息并由框架再次解码）访问远程服务。这在静态类型编程语言中非常有用，但在动态类型编程语言中很少（参阅“代码生成和动态类型化语言”）。

由于WSDL的设计不是人类可读的，而且由于SOAP消息通常是手动构建的过于复杂，所以SOAP的用户在很大程度上依赖于工具支持，代码生成和IDE【38】。对于SOAP供应商不支持的编程语言的用户来说，与SOAP服务的集成是困难的。

尽管SOAP及其各种扩展表面上是标准化的，但是不同厂商的实现之间的互操作性往往会造成问题【39】。由于所有这些原因，尽管许多大型企业仍然使用SOAP，但在大多数小公司中已经不再受到青睐。

REST风格的API倾向于更简单的方法，通常涉及较少的代码生成和自动化工具。定义格式（如OpenAPI，也称为Swagger 【40】）可用于描述RESTful API并生成文档。

远程过程调用（RPC）的问题

Web服务仅仅是通过网络进行API请求的一系列技术的最新版本，其中许多技术受到了大量的炒作，但是存在严重的问题。 Enterprise JavaBeans（EJB）和Java的远程方法调用（**RMI**）仅限于Java。分布式组件对象模型（**DCOM**）仅限于Microsoft平台。公共对象请求代理体系结构（**CORBA**）过于复杂，不提供前向或后向兼容性【41】。

所有这些都是基于远程过程调用（**RPC**）的思想，该过程调用自20世纪70年代以来一直存在【42】。 RPC模型试图向远程网络服务发出请求，看起来与在同一进程中调用编程语言中的函数或方法相同（这种抽象称为位置透明）。尽管RPC起初看起来很方便，但这种方法根本上是有缺陷的【43,44】。网络请求与本地函数调用非常不同：

- 本地函数调用是可预测的，并且成功或失败，这仅取决于受您控制的参数。网络请求是不可预知的：由于网络问题，请求或响应可能会丢失，或者远程计算机可能很慢或不可用，这些问题完全不在您的控制范围之内。网络问题是常见的，所以你必须预测他们，例如通过重试失败的请求。
- 本地函数调用要么返回结果，要么抛出异常，或者永远不返回（因为进入无限循环或进程崩溃）。网络请求有另一个可能的结果：由于超时，它可能会返回没有结果。在这种情况下，你根本不知道发生了什么：如果你没有得到来自远程服务的响应，你无法知道请求是否通过。 （我们将在第8章更详细地讨论这个问题。）
- 如果您重试失败的网络请求，可能会发生请求实际上正在通过，只有响应丢失。在这种情况下，重试将导致该操作被执行多次，除非您在协议中引入除重（幂等（**idempotence**））机制。本地函数调用没有这个问题。 （在第十一章更详细地讨论幂等性）
- 每次调用本地功能时，通常需要大致相同的时间来执行。网络请求比函数调用要慢得多，而且其延迟也是非常可变的：在不到一毫秒的时间内它可能会完成，但是当网络拥

塞或者远程服务超载时，可能需要几秒钟的时间完全一样的东西。
- 调用本地函数时，可以高效地将引用（指针）传递给本地内存中的对象。当你发出一个网络请求时，所有这些参数都需要被编码成可以通过网络发送的一系列字节。没关系，如果参数是像数字或字符串这样的基本类型，但是对于较大的对象很快就会变成问题。

客户端和服务可以用不同的编程语言实现，所以RPC框架必须将数据类型从一种语言翻译成另一种语言。这可能会捅出大篓子，因为不是所有的语言都具有相同的类型 —— 例如回想一下JavaScript的数字大于2^{53}的问题（参阅“JSON，XML和二进制变体”）。用单一语言编写的单个进程中不存在此问题。

所有这些因素意味着尝试使远程服务看起来像编程语言中的本地对象一样毫无意义，因为这是一个根本不同的事情。 REST的部分吸引力在于，它并不试图隐藏它是一个网络协议的事实（尽管这似乎并没有阻止人们在REST之上构建RPC库）。

RPC的当前方向

尽管有这样那样的问题，RPC不会消失。在本章提到的所有编码的基础上构建了各种RPC框架：例如，Thrift和Avro带有RPC支持，gRPC是使用Protocol Buffers的RPC实现，Finagle也使用Thrift，Rest.li使用JSON over HTTP。

这种新一代的RPC框架更加明确的是，远程请求与本地函数调用不同。例如，Finagle和Rest.li 使用futures（promises）来封装可能失败的异步操作。 `Futures` 还可以简化需要并行发出多项服务的情况，并将其结果合并【45】。 gRPC支持流，其中一个调用不仅包括一个请求和一个响应，还包括一系列的请求和响应【46】。

其中一些框架还提供服务发现，即允许客户端找出在哪个IP地址和端口号上可以找到特定的服务。我们将在“请求路由”中回到这个主题。

使用二进制编码格式的自定义RPC协议可以实现比通用的JSON over REST更好的性能。但是，RESTful API还有其他一些显着的优点：对于实验和调试（只需使用Web浏览器或命令行工具curl，无需任何代码生成或软件安装即可向其请求），它是受支持的所有的主流编程语言和平台，还有大量可用的工具（服务器，缓存，负载平衡器，代理，防火墙，监控，调试工具，测试工具等）的生态系统。由于这些原因，REST似乎是公共API的主要风格。 RPC框架的主要重点在于同一组织拥有的服务之间的请求，通常在同一数据中心内。

数据编码与RPC的演化

对于可演化性，重要的是可以独立更改和部署RPC客户端和服务器。与通过数据库流动的数据相比（如上一节所述），我们可以在通过服务进行数据流的情况下做一个简化的假设：假定所有的服务器都会先更新，其次是所有的客户端。因此，您只需要在请求上具有向后兼容性，并且对响应具有前向兼容性。

RPC方案的前后向兼容性属性从它使用的编码方式中继承

- Thrift，gRPC（Protobuf）和Avro RPC可以根据相应编码格式的兼容性规则进行演变。

- 在SOAP中，请求和响应是使用XML模式指定的。这些可以演变，但有一些微妙的陷阱【47】。
- RESTful API通常使用JSON（没有正式指定的模式）用于响应，以及用于请求的JSON或URI编码/表单编码的请求参数。添加可选的请求参数并向响应对象添加新的字段通常被认为是保持兼容性的改变。

由于RPC经常被用于跨越组织边界的通信，所以服务的兼容性变得更加困难，因此服务的提供者经常无法控制其客户，也不能强迫他们升级。因此，需要长期保持兼容性，也许是无限期的。如果需要进行兼容性更改，则服务提供商通常会并排维护多个版本的服务API。

关于API版本化应该如何工作（即，客户端如何指示它想要使用哪个版本的API）没有一致意见【48】）。对于RESTful API，常用的方法是在URL或HTTP Accept头中使用版本号。对于使用API密钥来标识特定客户端的服务，另一种选择是将客户端请求的API版本存储在服务器上，并允许通过单独的管理界面更新该版本选项【49】。

消息传递中的数据流

我们一直在研究从一个过程到另一个过程的编码数据流的不同方式。到目前为止，我们已经讨论了REST和RPC（其中一个进程通过网络向另一个进程发送请求并期望尽可能快的响应）以及数据库（一个进程写入编码数据，另一个进程在将来再次读取）。

在最后一节中，我们将简要介绍一下RPC和数据库之间的异步消息传递系统。它们与RPC类似，因为客户端的请求（通常称为消息）以低延迟传送到另一个进程。它们与数据库类似，不是通过直接的网络连接发送消息，而是通过称为消息代理（也称为消息队列或面向消息的中间件）的中介来临时存储消息。

与直接RPC相比，使用消息代理有几个优点：

- 如果收件人不可用或过载，可以充当缓冲区，从而提高系统的可靠性。
- 它可以自动将消息重新发送到已经崩溃的进程，从而防止消息丢失。
- 避免发件人需要知道收件人的IP地址和端口号（这在虚拟机经常出入的云部署中特别有用）。
- 它允许将一条消息发送给多个收件人。
- 将发件人与收件人逻辑分离（发件人只是发布邮件，不关心使用者）。

然而，与RPC相比，差异在于消息传递通信通常是单向的：发送者通常不期望收到其消息的回复。一个进程可能发送一个响应，但这通常是在一个单独的通道上完成的。这种通信模式是异步的：发送者不会等待消息被传递，而只是发送它，然后忘记它。

消息掮客

过去，信息掮客主要是TIBCO，IBM WebSphere和webMethods等公司的商业软件的秀场。最近像RabbitMQ，ActiveMQ，HornetQ，NATS和Apache Kafka这样的开源实现已经流行起来。我们将在第11章中对它们进行更详细的比较。

详细的交付语义因实现和配置而异，但通常情况下，消息代理的使用方式如下：一个进程将消息发送到指定的队列或主题，代理确保将消息传递给一个或多个消费者或订阅者到那个队列或主题。在同一主题上可以有许多生产者和许多消费者。

一个主题只提供单向数据流。但是，消费者本身可能会将消息发布到另一个主题上（因此，可以将它们链接在一起，就像我们将在第11章中看到的那样），或者发送给原始消息的发送者使用的回复队列（允许请求/响应数据流，类似于RPC）。

消息代理通常不会执行任何特定的数据模型 - 消息只是包含一些元数据的字节序列，因此您可以使用任何编码格式。如果编码是向后兼容的，则您可以灵活地更改发行商和消费者的独立编码，并以任意顺序进行部署。

如果消费者重新发布消息到另一个主题，则可能需要小心保留未知字段，以防止前面在数据库环境中描述的问题（图4-7）。

分布式的Actor框架

Actor模型是单个进程中并发的编程模型。逻辑被封装在角色中，而不是直接处理线程（以及竞争条件，锁定和死锁的相关问题）。每个角色通常代表一个客户或实体，它可能有一些本地状态（不与其他任何角色共享），它通过发送和接收异步消息与其他角色通信。消息传送不保证：在某些错误情况下，消息将丢失。由于每个角色一次只能处理一条消息，因此不需要担心线程，每个角色可以由框架独立调度。

在分布式的行为者框架中，这个编程模型被用来跨越多个节点来扩展应用程序。不管发送方和接收方是在同一个节点上还是在不同的节点上，都使用相同的消息传递机制。如果它们在不同的节点上，则该消息被透明地编码成字节序列，通过网络发送，并在另一侧解码。

位置透明在actor模型中比在RPC中效果更好，因为actor模型已经假定消息可能会丢失，即使在单个进程中也是如此。尽管网络上的延迟可能比同一个进程中的延迟更高，但是在使用参与者模型时，本地和远程通信之间的基本不匹配是较少的。

分布式的Actor框架实质上是将消息代理和角色编程模型集成到一个框架中。但是，如果要执行基于角色的应用程序的滚动升级，则仍然需要担心向前和向后兼容性问题，因为消息可能会从运行新版本的节点发送到运行旧版本的节点，反之亦然。

三个流行的分布式actor框架处理消息编码如下：

- 默认情况下，Akka使用Java的内置序列化，不提供前向或后向兼容性。 但是，你可以用类似缓冲区的东西替代它，从而获得滚动升级的能力【50】。
- Orleans 默认使用不支持滚动升级部署的自定义数据编码格式; 要部署新版本的应用程序，您需要设置一个新的群集，将流量从旧群集迁移到新群集，然后关闭旧群集【51,52】。 像Akka一样，可以使用自定义序列化插件。
- 在Erlang OTP中，对记录模式进行更改是非常困难的（尽管系统具有许多为高可用性设计的功能）。 滚动升级是可能的，但需要仔细计划【53】。 一个新的实验性的 `maps` 数据类型（2014年在Erlang R17中引入的类似于JSON的结构）可能使得这个数据类型在未

来更容易【54】。

本章小结

在本章中，我们研究了将数据结构转换为网络中的字节或磁盘上的字节的几种方法。我们看到了这些编码的细节不仅影响其效率，更重要的是应用程序的体系结构和部署它们的选项。

特别是，许多服务需要支持滚动升级，其中新版本的服务逐步部署到少数节点，而不是同时部署到所有节点。滚动升级允许在不停机的情况下发布新版本的服务（从而鼓励在罕见的大型版本上频繁发布小型版本），并使部署风险降低（允许在影响大量用户之前检测并回滚有故障的版本）。这些属性对于可演化性，以及对应用程序进行更改的容易性都是非常有利的。

在滚动升级期间，或出于各种其他原因，我们必须假设不同的节点正在运行我们的应用程序代码的不同版本。因此，在系统周围流动的所有数据都是以提供向后兼容性（新代码可以读取旧数据）和向前兼容性（旧代码可以读取新数据）的方式进行编码是重要的。

我们讨论了几种数据编码格式及其兼容性属性：

- 编程语言特定的编码仅限于单一编程语言，并且往往无法提供前向和后向兼容性。
- JSON，XML和CSV等文本格式非常普遍，其兼容性取决于您如何使用它们。他们有可选的模式语言，这有时是有用的，有时是一个障碍。这些格式对于数据类型有些模糊，所以你必须小心数字和二进制字符串。
- 像Thrift，Protocol Buffers和Avro这样的二进制模式驱动格式允许使用清晰定义的前向和后向兼容性语义进行紧凑，高效的编码。这些模式可以用于静态类型语言的文档和代码生成。但是，他们有一个缺点，就是在数据可读之前需要对数据进行解码。

我们还讨论了数据流的几种模式，说明了数据编码是重要的不同场景：

- 数据库，写入数据库的进程对数据进行编码，并从数据库读取进程对其进行解码
- RPC和REST API，客户端对请求进行编码，服务器对请求进行解码并对响应进行编码，客户端最终对响应进行解码
- 异步消息传递（使用消息代理或参与者），其中节点之间通过发送消息进行通信，消息由发送者编码并由接收者解码

我们可以小心地得出这样的结论：前向兼容性和滚动升级在某种程度上是可以实现的。愿您的应用程序的演变迅速、敏捷部署。

参考文献

1. “Java Object Serialization Specification,” *docs.oracle.com*, 2010.
2. “Ruby 2.2.0 API Documentation,” *ruby-doc.org*, Dec 2014.

3. “The Python 3.4.3 Standard Library Reference Manual,” *docs.python.org*, February 2015.
4. “EsotericSoftware/kryo,” *github.com*, October 2014.
5. “CWE-502: Deserialization of Untrusted Data,” Common Weakness Enumeration, *cwe.mitre.org*, July 30, 2014.
6. Steve Breen: “What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability,” *foxglovesecurity.com*, November 6, 2015.
7. Patrick McKenzie: “What the Rails Security Issue Means for Your Startup,” *kalzumeus.com*, January 31, 2013.
8. Eishay Smith: “jvm-serializers wiki,” *github.com*, November 2014.
9. “XML Is a Poor Copy of S-Expressions,” *c2.com* wiki.
10. Matt Harris: “Snowflake: An Update and Some Very Important Information,” email to *Twitter Development Talk* mailing list, October 19, 2010.
11. Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry S. Thompson: “XML Schema 1.1,” W3C Recommendation, May 2001.
12. Francis Galiegue, Kris Zyp, and Gary Court: “JSON Schema,” IETF Internet-Draft, February 2013.
13. Yakov Shafranovich: “RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files,” October 2005.
14. “MessagePack Specification,” *msgpack.org*. Mark Slee, Aditya Agarwal, and Marc Kwiatkowski: “Thrift: Scalable Cross-Language Services Implementation,” Facebook technical report, April 2007.
15. “Protocol Buffers Developer Guide,” Google, Inc., *developers.google.com*.
16. Igor Anishchenko: “Thrift vs Protocol Buffers vs Avro - Biased Comparison,” *slideshare.net*, September 17, 2012.
17. “A Matrix of the Features Each Individual Language Library Supports,” *wiki.apache.org*.
18. Martin Kleppmann: “Schema Evolution in Avro, Protocol Buffers and Thrift,” *martin.kleppmann.com*, December 5, 2012.
19. “Apache Avro 1.7.7 Documentation,” *avro.apache.org*, July 2014.

20. Doug Cutting, Chad Walters, Jim Kellerman, et al.: "[PROPOSAL] New Subproject: Avro," email thread on *hadoop-general* mailing list, *mail-archives.apache.org*, April 2009.
21. Tony Hoare: "Null References: The Billion Dollar Mistake," at *QCon London*, March 2009.
22. Aditya Auradkar and Tom Quiggle: "Introducing Espresso—LinkedIn's Hot New Distributed Document Store," *engineering.linkedin.com*, January 21, 2015.
23. Jay Kreps: "Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform (Part 2)," *blog.confluent.io*, February 25, 2015.
24. Gwen Shapira: "The Problem of Managing Schemas," *radar.oreilly.com*, November 4, 2014.
25. "Apache Pig 0.14.0 Documentation," *pig.apache.org*, November 2014.
26. John Larmouth: *ASN.1Complete*. Morgan Kaufmann, 1999. ISBN: 978-0-122-33435-1
27. Russell Housley, Warwick Ford, Tim Polk, and David Solo: "RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile," IETF Network Working Group, Standards Track, January 1999.
28. Lev Walkin: "Question: Extensibility and Dropping Fields," *lionet.info*, September 21, 2010.
29. Jesse James Garrett: "Ajax: A New Approach to Web Applications," *adaptivepath.com*, February 18, 2005.
30. Sam Newman: *Building Microservices*. O'Reilly Media, 2015. ISBN: 978-1-491-95035-7
31. Chris Richardson: "Microservices: Decomposing Applications for Deployability and Scalability," *infoq.com*, May 25, 2014.
32. Pat Helland: "Data on the Outside Versus Data on the Inside," at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.
33. Roy Thomas Fielding: "Architectural Styles and the Design of Network-Based Software Architectures," PhD Thesis, University of California, Irvine, 2000.
34. Roy Thomas Fielding: "REST APIs Must Be Hypertext-Driven," *roy.gbiv.com*, October 20 2008.
35. "REST in Peace, SOAP," *royal.pingdom.com*, October 15, 2010.
36. "Web Services Standards as of Q1 2007," *innoq.com*, February 2007.

37. Pete Lacey: "The S Stands for Simple," *harmful.cat-v.org*, November 15, 2006.

38. Stefan Tilkov: "Interview: Pete Lacey Criticizes Web Services," *infoq.com*, December 12, 2006.

39. "OpenAPI Specification (fka Swagger RESTful API Documentation Specification) Version 2.0," *swagger.io*, September 8, 2014.

40. Michi Henning: "The Rise and Fall of CORBA," *ACM Queue*, volume 4, number 5, pages 28–34, June 2006. doi:10.1145/1142031.1142044

41. Andrew D. Birrell and Bruce Jay Nelson: "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* (TOCS), volume 2, number 1, pages 39–59, February 1984. doi:10.1145/2080.357392

42. Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall: "A Note on Distributed Computing," Sun Microsystems Laboratories, Inc., Technical Report TR-94-29, November 1994.

43. Steve Vinoski: "Convenience over Correctness," *IEEE Internet Computing*, volume 12, number 4, pages 89–92, July 2008. doi:10.1109/MIC.2008.75

44. Marius Eriksen: "Your Server as a Function," at *7th Workshop on Programming Languages and Operating Systems* (PLOS), November 2013. doi:10.1145/2525528.2525538

45. "grpc-common Documentation," Google, Inc., *github.com*, February 2015.

46. Aditya Narayan and Irina Singh: "Designing and Versioning Compatible Web Services," *ibm.com*, March 28, 2007.

47. Troy Hunt: "Your API Versioning Is Wrong, Which Is Why I Decided to Do It 3 Different Wrong Ways," *troyhunt.com*, February 10, 2014.

48. "API Upgrades," Stripe, Inc., April 2015.

49. Jonas Bonér: "Upgrade in an Akka Cluster," email to *akka-user* mailing list, *grokbase.com*, August 28, 2013.

50. Philip A. Bernstein, Sergey Bykov, Alan Geller, et al.: "Orleans: Distributed Virtual Actors for Programmability and Scalability," Microsoft Research Technical Report MSR-TR-2014-41, March 2014.

51. "Microsoft Project Orleans Documentation," Microsoft Research, *dotnet.github.io*, 2015.

52. David Mercer, Sean Hinde, Yinso Chen, and Richard A O'Keefe: “beginner: Updating Data Structures,” email thread on *erlang-questions* mailing list, *erlang.com*, October 29, 2007.

53. Fred Hebert: “Postscript: Maps,” *learnyousomeerlang.com*, April 9, 2014.

上一章	目录	下一章
第三章：存储与检索	设计数据密集型应用	第二部分：分布式数据

第二部分：分布式数据

> 一个成功的技术，现实的优先级必须高于公关，你可以糊弄别人，但糊弄不了自然规律。
>
> ——罗杰斯委员会报告（1986）

在本书的第一部分中，我们讨论了数据系统的各个方面，但仅限于数据存储在单台机器上的情况。现在我们到了第二部分，进入更高的层次，并提出一个问题：如果多台机器参与数据的存储和检索，会发生什么？

你可能会出于各种各样的原因，希望将数据库分布到多台机器上：

可扩展性

如果你的数据量、读取负载、写入负载超出单台机器的处理能力，可以将负载分散到多台计算机上。

容错/高可用性

如果你的应用需要在单台机器（或多台机器，网络或整个数据中心）出现故障的情况下仍然能继续工作，则可使用多台机器，以提供冗余。一台故障时，另一台可以接管。

延迟

如果在世界各地都有用户，你也许会考虑在全球范围部署多个服务器，从而每个用户可以从地理上最近的数据中心获取服务，避免了等待网络数据包穿越半个世界。

扩展至更高的载荷

如果你需要的只是扩展至更高的载荷（**load**），最简单的方法就是购买更强大的机器（有时称为垂直扩展（**vertical scaling**）或向上扩展（**scale up**））。许多处理器，内存和磁盘可以在同一个操作系统下相互连接，快速的相互连接允许任意处理器访问内存或磁盘的任意部分。在这种共享内存架构（**shared-memory architecture**）中，所有的组件都可以看作一台单独的机器。

> i. 在大型机中，尽管任意处理器都可以访问内存的任意部分，但总有一些内存区域与一些处理器更接近（称为非均匀内存访问（**nonuniform memory access, NUMA**）【1】）。为了有效利用这种架构特性，需要对处理进行细分，以便每个处理器主要访问临近的内存，这意味着即使表面上看起来只有一台机器在运行，分区（**partitioning**）仍然是必要的。↩

共享内存方法的问题在于，成本增长速度快于线性增长：一台有着双倍处理器数量，双倍内存大小，双倍磁盘容量的机器，通常成本会远远超过原来的两倍。而且可能因为存在瓶颈，并不足以处理双倍的载荷。

共享内存架构可以提供有限的容错能力，高端机器可以使用热插拔的组件（不关机更换磁盘，内存模块，甚至处理器）——但它必然囿于单个地理位置的桎梏。

另一种方法是共享磁盘架构（**shared-disk architecture**），它使用多台具有独立处理器和内存的机器，但将数据存储在机器之间共享的磁盘阵列上，这些磁盘通过快速网络连接[ii]。这种架构用于某些数据仓库，但竞争和锁定的开销限制了共享磁盘方法的可扩展性【2】。

> ii. 网络附属存储（Network Attached Storage, NAS），或存储区网络（**Storage Area Network, SAN**） ↩

无共享架构

相比之下，无共享架构（**shared-nothing architecture**）（有时称为水平扩展（**horizontal scale**） 或向外扩展（**scale out**））已经相当普及。在这种架构中，运行数据库软件的每台机器/虚拟机都称为节点（**node**）。每个节点只使用各自的处理器，内存和磁盘。节点之间的任何协调，都是在软件层面使用传统网络实现的。

无共享系统不需要使用特殊的硬件，所以你可以用任意机器——比如性价比最好的机器。你也许可以跨多个地理区域分布数据从而减少用户延迟，或者在损失一整个数据中心的情况下幸免于难。随着云端虚拟机部署的出现，即使是小公司，现在无需Google级别的运维，也可以实现异地分布式架构。

在这一部分里，我们将重点放在无共享架构上。它不见得是所有场景的最佳选择，但它是最需要你谨慎从事的架构。如果你的数据分布在多个节点上，你需要意识到这样一个分布式系统中约束和权衡 ——数据库并不能魔术般地把这些东西隐藏起来。

虽然分布式无共享架构有许多优点，但它通常也会给应用带来额外的复杂度，有时也会限制你可用数据模型的表达力。在某些情况下，一个简单的单线程程序可以比一个拥有超过100个CPU核的集群表现得更好【4】。另一方面，无共享系统可以非常强大。接下来的几章，将详细讨论分布式数据会带来的问题。

复制 vs 分区

数据分布在多个节点上有两种常见的方式：

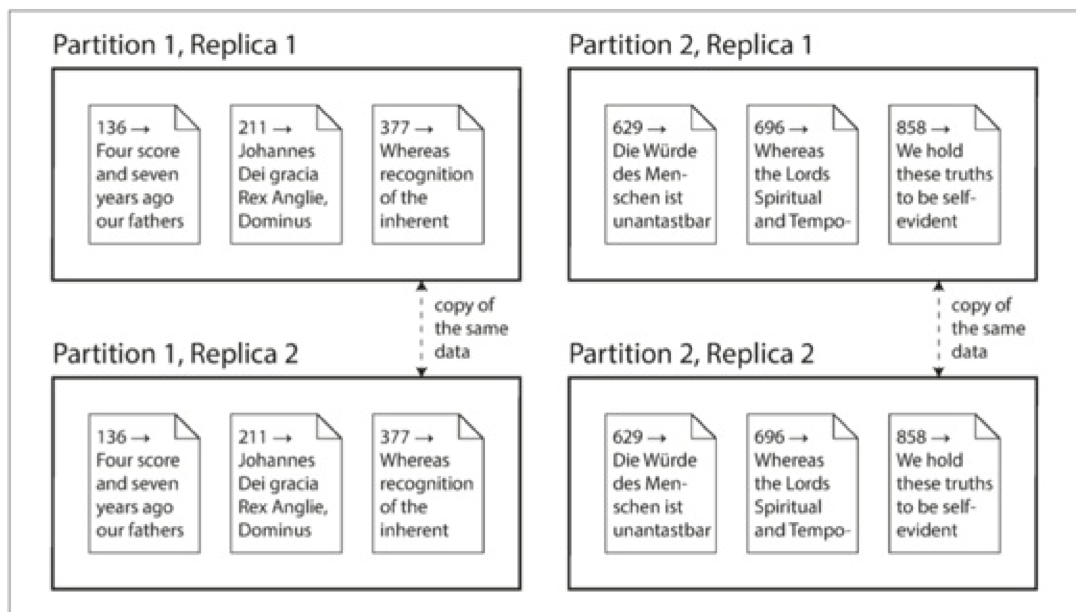
复制（***Replication***）

在几个不同的节点上保存数据的相同副本，可能放在不同的位置。 复制提供了冗余：如果一些节点不可用，剩余的节点仍然可以提供数据服务。 复制也有助于改善性能。 第五章将讨论复制。

分区 ***(Partitioning)***

将一个大型数据库拆分成较小的子集（称为分区（**partitions**））,从而不同的分区可以指派给不同的节点（**node**）（亦称分片（**shard**））。 第六章将讨论分区。

复制和分区是不同的机制，但它们经常同时使用。如图II-1所示。



图**II-1** 一个数据库切分为两个分区，每个分区都有两个副本

理解了这些概念，就可以开始讨论在分布式系统中需要做出的困难抉择。第七章将讨论事务**(Transaction)**，这对于了解数据系统中可能出现的各种问题，以及我们可以做些什么很有帮助。第八章和第九章将讨论分布式系统的根本局限性。

在本书的第三部分中，将讨论如何将多个（可能是分布式的）数据存储集成为一个更大的系统，以满足复杂的应用需求。 但首先，我们来聊聊分布式的数据。

索引

参考文献

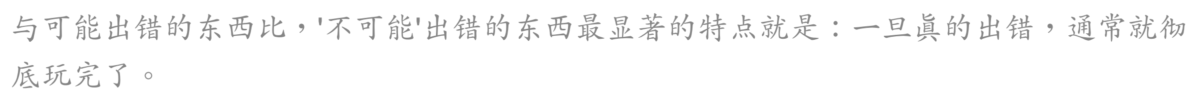
1. Ulrich Drepper: “What Every Programmer Should Know About Memory,” akka-dia.org, November 21, 2007.

2. Ben Stopford: “Shared Nothing vs. Shared Disk Architectures: An Independent View,” benstopford.com, November 24, 2009.

1. Michael Stonebraker: “The Case for Shared Nothing,” IEEE Database EngineeringBulletin, volume 9, number 1, pages 4–9, March 1986.
2. Frank McSherry, Michael Isard, and Derek G. Murray: “Scalability! But at What COST?,” at 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS),May 2015.

上一章	目录	下一章
第四章：编码与演化	设计数据密集型应用	第五章：复制

5. 复制



> 与可能出错的东西比，'不可能'出错的东西最显著的特点就是：一旦真的出错，通常就彻底玩完了。
>
> ——道格拉斯·亚当斯（1992）

[TOC]

复制意味着在通过网络连接的多台机器上保留相同数据的副本。正如在第二部分简介中所讨论的那样，我们希望能复制数据，可能出于各种各样的原因：

- 使得数据与用户在地理上接近（从而减少延迟）
- 即使系统的一部分出现故障，系统也能继续工作（从而提高可用性）
- 扩展可以接受读请求的机器数量（从而提高读取吞吐量）

本章将假设你的数据集非常小，每台机器都可以保存整个数据集的副本。在第6章中将放宽这个假设，讨论对单个机器来说太大的数据集的分割（分片）。在后面的章节中，我们将讨论复制数据系统中可能发生的各种故障，以及如何处理这些故障。

如果复制中的数据不会随时间而改变，那复制就很简单：将数据复制到每个节点一次就万事大吉。复制的困难之处在于处理复制数据的变更（**change**），这就是本章所要讲的。我们将讨论三种流行的变更复制算法：单领导者（**single leader**），多领导者（**multi leader**）和无领导者（**leaderless**）。几乎所有分布式数据库都使用这三种方法之一。

在复制时需要进行许多权衡：例如，使用同步复制还是异步复制？如何处理失败的副本？这些通常是数据库中的配置选项，细节因数据库而异，但原理在许多不同的实现中都类似。本章会讨论这些决策的后果。

数据库的复制算得上是老生常谈了 ——70年代研究得出的基本原则至今没有太大变化【1】，因为网络的基本约束仍保持不变。然而在研究之外，许多开发人员仍然假设一个数据库只有一个节点。分布式数据库变为主流只是最近发生的事。许多程序员都是这一领域的新手，因此对于诸如最终一致性（**eventual consistency**）等问题存在许多误解。在“复制延迟问题”一节，我们将更加精确地了解最终的一致性，并讨论诸如读己之写（**read-your-writes**）和单调读（**monotonic read**）保证等内容。

领导者与追随者

存储数据库副本的每个节点称为副本（**replica**）。当存在多个副本时，会不可避免的出现一个问题：如何确保所有数据都落在了所有的副本上？

每一次向数据库的写入操作都需要传播到所有副本上，否则副本就会包含不一样的数据。最常见的解决方案被称为基于领导者的复制（**leader-based replication**）（也称主动/被动（**active/passive**） 或 主/从（**master/slave**）复制），如图5-1所示。它的工作原理如下：

1. 副本之一被指定为领导者（**leader**），也称为 主库（**master**） ，首要（**primary**）。当客户端要向数据库写入时，它必须将请求发送给领导者，领导者会将新数据写入其本地存储。
2. 其他副本被称为追随者（**followers**），亦称为只读副本（**read replicas**），从库（**slaves**），次要（ **sencondaries**），热备（**hot-standby**）[i]。每当领导者将新数据写入本地存储时，它也会将数据变更发送给所有的追随者，称之为复制日志（**replication**

log）记录或变更流（**change stream**）。每个跟随者从领导者拉取日志，并相应更新其本地数据库副本，方法是按照领导者处理的相同顺序应用所有写入。

3. 当客户想要从数据库中读取数据时，它可以向领导者或追随者查询。但只有领导者才能接受写操作（从客户端的角度来看从库都是只读的）。

> i. 不同的人对热（**hot**），温（**warn**），冷（**cold**）备份服务器有不同的定义。例如在PostgreSQL中，热备（**hot standby**）指的是能接受客户端读请求的副本。而温备（**warm standby**）只是追随领导者，但不处理客户端的任何查询。就本书而言，这些差异并不重要。↩

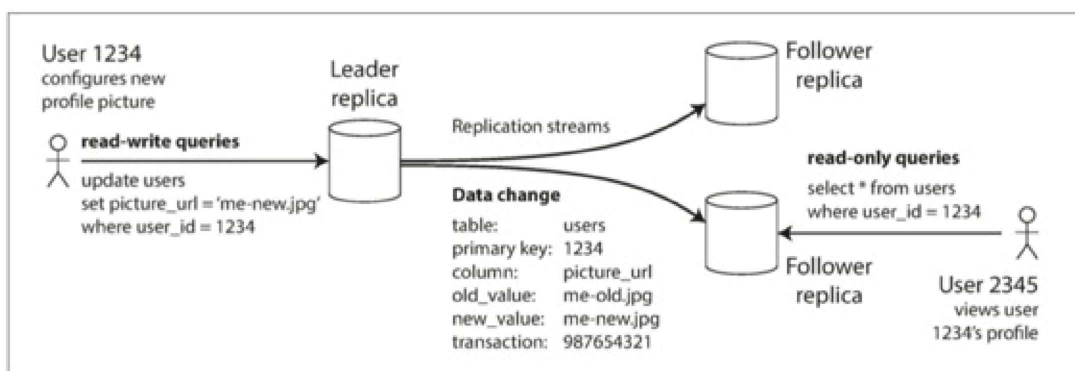


图**5-1** 基于领导者**(**主**-**从**)**的复制

这种复制模式是许多关系数据库的内置功能，如PostgreSQL（从9.0版本开始），MySQL，Oracle Data Guard 【2】和SQL Server的AlwaysOn可用性组【3】。它也被用于一些非关系数据库，包括MongoDB，RethinkDB和Espresso 【4】。最后，基于领导者的复制并不仅限于数据库：像Kafka 【5】和RabbitMQ高可用队列【6】这样的分布式消息代理也使用它。某些网络文件系统，例如DRBD这样的块复制设备也与之类似。

同步复制与异步复制

复制系统的一个重要细节是：复制是同步（**synchronously**）发生还是异步（**asynchronously**）发生。（在关系型数据库中这通常是一个配置项，其他系统通常硬编码为其中一个）。

想象图5-1中发生的情况，网站的用户更新他们的个人头像。在某个时间点，客户向主库发送更新请求；不久之后主库就收到了请求。在某个时刻，主库又会将数据变更转发给自己的从库。最后，主库通知客户更新成功。

图5-2显示了系统各个组件之间的通信：用户客户端，主库和两个从库。时间从左到右流动。请求或响应消息用粗箭头表示。

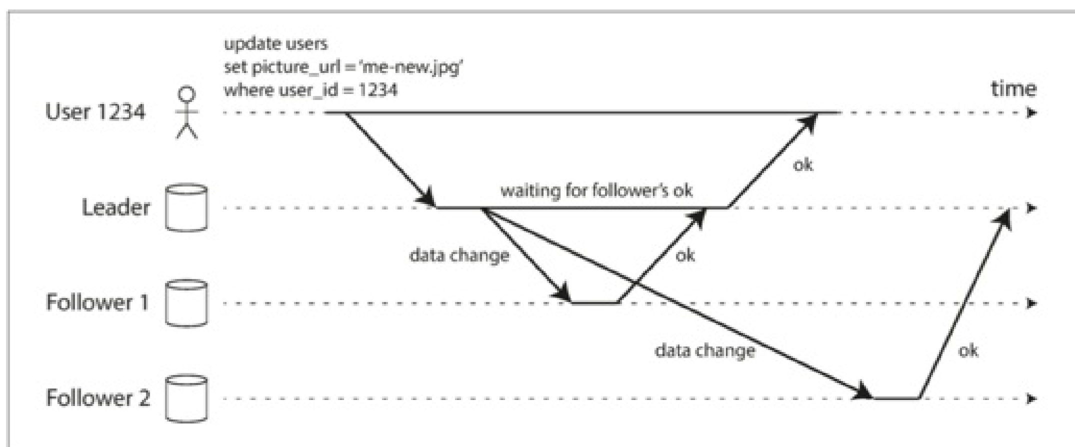


图**5-2** 基于领导者的复制：一个同步从库和一个异步从库

在图5-2的示例中，从库1的复制是同步的：在向用户报告写入成功，并使结果对其他用户可见之前，主库需要等待从库1的确认，确保从库1已经收到写入操作。以及在使写入对其他客户端可见之前接收到写入。跟随者2的复制是异步的：主库发送消息，但不等待从库的响应。

在这幅图中，从库2处理消息前存在一个显着的延迟。通常情况下，复制的速度相当快：大多数数据库系统能在一秒向从库应用变更，但它们不能提供复制用时的保证。有些情况下，从库可能落后主库几分钟或更久；例如：从库正在从故障中恢复，系统在最大容量附近运行，或者如果节点间存在网络问题。

同步复制的优点是，从库保证有与主库一致的最新数据副本。如果主库突然失效，我们可以确信这些数据仍然能在从库上上找到。缺点是，如果同步从库没有响应（比如它已经崩溃，或者出现网络故障，或其它任何原因），主库就无法处理写入操作。主库必须阻止所有写入，并等待同步副本再次可用。

因此，将所有从库都设置为同步的是不切实际的：任何一个节点的中断都会导致整个系统停滞不前。实际上，如果在数据库上启用同步复制，通常意味着其中一个跟随者是同步的，而其他的则是异步的。如果同步从库变得不可用或缓慢，则使一个异步从库同步。这保证你至少在两个节点上拥有最新的数据副本：主库和同步从库。 这种配置有时也被称为半同步（**semi-synchronous**）【7】。

通常情况下，基于领导者的复制都配置为完全异步。 在这种情况下，如果主库失效且不可恢复，则任何尚未复制给从库的写入都会丢失。 这意味着即使已经向客户端确认成功，写入也不能保证持久（**Durable**）。 然而，一个完全异步的配置也有优点：即使所有的从库都落后了，主库也可以继续处理写入。

弱化的持久性可能听起来像是一个坏的折衷，无论如何，异步复制已经被广泛使用了，特别当有很多追随者，或追随者异地分布时。 稍后将在“复制延迟问题”中回到这个问题。

> ### 关于复制的研究
>
> 对于异步复制系统而言，主库故障时有可能丢失数据。这可能是一个严重的问题，因此研究人员仍在研究不丢数据但仍能提供良好性能和可用性的复制方法。 例如，链式复制【8,9】]是同步复制的一种变体，已经在一些系统（如Microsoft Azure存储【10,11】）中成功实现。
>
> 复制的一致性与共识（**consensus**）（使几个节点就某个值达成一致）之间有着密切的联系，第9章将详细地探讨这一领域的理论。本章主要讨论实践中数据库常用的简单复制形式。

设置新从库

有时候需要设置一个新的从库：也许是为了增加副本的数量，或替换失败的节点。如何确保新的从库拥有主库数据的精确副本？

简单地将数据文件从一个节点复制到另一个节点通常是不够的：客户端不断向数据库写入数据，数据总是在不断变化，标准的数据副本会在不同的时间点总是不一样。复制的结果可能没有任何意义。

可以通过锁定数据库（使其不可用于写入）来使磁盘上的文件保持一致，但是这会违背高可用的目标。幸运的是，拉起新的从库通常并不需要停机。从概念上讲，过程如下所示：

1. 在某个时刻获取主库的一致性快照（如果可能），而不必锁定整个数据库。大多数数据库都具有这个功能，因为它是备份必需的。对于某些场景，可能需要第三方工具，例如MySQL的innobackupex 【12】。
2. 将快照复制到新的从库节点。
3. 从库连接到主库，并拉取快照之后发生的所有数据变更。这要求快照与主库复制日志中的位置精确关联。该位置有不同的名称：例如，PostgreSQL将其称为日志序列号（**log sequence number, LSN**），MySQL将其称为二进制日志坐标（**binlog coordinates**）。
4. 当从库处理完快照之后积压的数据变更，我们说它赶上（**caught up**）了主库。现在它可以继续处理主库产生的数据变化了。

建立从库的实际步骤因数据库而异。在某些系统中，这个过程是完全自动化的，而在另外一些系统中，它可能是一个需要由管理员手动执行的，有点神秘的多步骤工作流。

处理节点宕机

系统中的任何节点都可能宕机，可能因为意外的故障，也可能由于计划内的维护（例如，重启机器以安装内核安全补丁）。对运维而言，能在系统不中断服务的情况下重启单个节点好处多多。我们的目标是，即使个别节点失效，也能保持整个系统运行，并尽可能控制节点停机带来的影响。

如何通过基于主库的复制实现高可用？

从库失效：追赶恢复

在其本地磁盘上，每个从库记录从主库收到的数据变更。如果从库崩溃并重新启动，或者，如果主库和从库之间的网络暂时中断，则比较容易恢复：从库可以从日志中知道，在发生故障之前处理的最后一个事务。因此，从库可以连接到主库，并请求在从库断开连接时发生的所有数据变更。当应用完所有这些变化后，它就赶上了主库，并可以像以前一样继续接收数据变更流。

主库失效：故障转移

主库失效处理起来相当棘手：其中一个从库需要被提升为新的主库，需要重新配置客户端，以将它们的写操作发送给新的主库，其他从库需要开始拉取来自新主库的数据变更。这个过程被称为故障转移（**failover**）。

故障转移可以手动进行（通知管理员主库挂了，并采取必要的步骤来创建新的主库）或自动进行。自动故障转移过程通常由以下步骤组成：

1. 确认主库失效。有很多事情可能会出错：崩溃，停电，网络问题等等。没有万无一失的方法来检测出现了什么问题，所以大多数系统只是简单使用超时（**Timeout**）：节点频繁地相互来回传递消息，并且如果一个节点在一段时间内（例如30秒）没有响应，就认为它挂了（因为计划内维护而故意关闭主库不算）。
2. 选择一个新的主库。这可以通过选举过程（主库由剩余副本以多数选举产生）来完成，或者可以由之前选定的控制器节点（**controller node**）来指定新的主库。主库的最佳人选通常是拥有旧主库最新数据副本的从库（最小化数据损失）。让所有的节点同意一个新的领导者，是一个共识问题，将在第9章详细讨论。
3. 重新配置系统以启用新的主库。客户端现在需要将它们的写请求发送给新主库（将在“请求路由”中讨论这个问题）。如果老领导回来，可能仍然认为自己是主库，没有意识到其他副本已经让它下台了。系统需要确保老领导认可新领导，成为一个从库。

故障转移会出现很多大麻烦：

- 如果使用异步复制，则新主库可能没有收到老主库宕机前最后的写入操作。在选出新主库后，如果老主库重新加入集群，新主库在此期间可能会收到冲突的写入，那这些写入该如何处理？最常见的解决方案是简单丢弃老主库未复制的写入，这很可能打破客户对于数据持久性的期望。

- 如果数据库需要和其他外部存储相协调，那么丢弃写入内容是极其危险的操作。例如在GitHub 【13】的一场事故中，一个过时的MySQL从库被提升为主库。数据库使用自增ID作为主键，因为新主库的计数器落后于老主库的计数器，所以新主库重新分配了一些已经被老主库分配掉的ID作为主键。这些主键也在Redis中使用，主键重用使得MySQL和Redis中数据产生不一致，最后导致一些私有数据泄漏到错误的用户手中。

- 发生某些故障时（见第8章）可能会出现两个节点都以为自己是主库的情况。这种情况称为脑裂(**split brain**)，非常危险：如果两个主库都可以接受写操作，却没有冲突解决机制（参见“多领导者复制”），那么数据就可能丢失或损坏。一些系统采取了安全防范措施：当检测到两个主库节点同时存在时会关闭其中一个节点[ii]，但设计粗糙的机制可能最后会导致两个节点都被关闭【14】。

> ii. 这种机制称为屏蔽（**fencing**），充满感情的术语是：爆彼之头（**Shoot The Other Node In The Head, STONITH**）。↩

- 主库被宣告死亡之前的正确超时应该怎么配置？在主库失效的情况下，超时时间越长，意味着恢复时间也越长。但是如果超时设置太短，又可能会出现不必要的故障转移。例如，临时负载峰值可能导致节点的响应时间超时，或网络故障可能导致数据包延迟。如果系统已经处于高负载或网络问题的困扰之中，那么不必要的故障切换可能会让情况变得更糟糕。

这些问题没有简单的解决方案。因此，即使软件支持自动故障切换，不少运维团队还是更愿意手动执行故障转移。

节点故障、不可靠的网络、对副本一致性，持久性，可用性和延迟的权衡 ，这些问题实际上是分布式系统中的基本问题。第8章和第9章将更深入地讨论它们。

复制日志的实现

基于主库的复制底层是如何工作的？实践中有好几种不同的复制方式，所以先简要地看一下。

基于语句的复制

在最简单的情况下，主库记录下它执行的每个写入请求（语句（**statement**））并将该语句日志发送给其从库。对于关系数据库来说，这意味着每个`INSERT`，`UPDATE`或`DELETE`语句都被转发给每个从库，每个从库解析并执行该SQL语句，就像从客户端收到一样。

虽然听上去很合理，但有很多问题会搞砸这种复制方式：

- 任何调用非确定性函数（**nondeterministic**）的语句，可能会在每个副本上生成不同的值。例如，使用`NOW()`获取当前日期时间，或使用`RAND()`获取一个随机数。
- 如果语句使用了自增列（**auto increment**），或者依赖于数据库中的现有数据（例如，`UPDATE ... WHERE <某些条件>`），则必须在每个副本上按照完全相同的顺序执行它们，否则可能会产生不同的效果。当有多个并发执行的事务时，这可能成为一个限制。
- 有副作用的语句（例如，触发器，存储过程，用户定义的函数）可能会在每个副本上产生不同的副作用，除非副作用是绝对确定的。

的确有办法绕开这些问题 ——例如，当语句被记录时，主库可以用固定的返回值替换任何不确定的函数调用，以便从库获得相同的值。但是由于边缘情况实在太多了，现在通常会选择其他的复制方法。

基于语句的复制在5.1版本前的MySQL中使用。因为它相当紧凑，现在有时候也还在用。但现在在默认情况下，如果语句中存在任何不确定性，MySQL会切换到基于行的复制（稍后讨论）。 VoltDB使用了基于语句的复制，但要求事务必须是确定性的，以此来保证安全【15】。

传输预写式日志（WAL）

在第3章中，我们讨论了存储引擎如何在磁盘上表示数据，并且我们发现，通常写操作都是追加到日志中：

- 对于日志结构存储引擎（请参阅“SSTables和LSM树”），日志是主要的存储位置。日志段在后台压缩，并进行垃圾回收。
- 对于覆写单个磁盘块的B树，每次修改都会先写入预写式日志（**Write Ahead Log, WAL**），以便崩溃后索引可以恢复到一个一致的状态。

在任何一种情况下，日志都是包含所有数据库写入的仅追加字节序列。可以使用完全相同的日志在另一个节点上构建副本：除了将日志写入磁盘之外，主库还可以通过网络将其发送给其从库。

当从库应用这个日志时，它会建立和主库一模一样数据结构的副本。

PostgreSQL和Oracle等使用这种复制方法【16】。主要缺点是日志记录的数据非常底层：WAL包含哪些磁盘块中的哪些字节发生了更改。这使复制与存储引擎紧密耦合。如果数据库将其存储格式从一个版本更改为另一个版本，通常不可能在主库和从库上运行不同版本的数据库软件。

看上去这可能只是一个微小的实现细节，但却可能对运维产生巨大的影响。如果复制协议允许从库使用比主库更新的软件版本，则可以先升级从库，然后执行故障转移，使升级后的节点之一成为新的主库，从而执行数据库软件的零停机升级。如果复制协议不允许版本不匹配（传输WAL经常出现这种情况），则此类升级需要停机。

逻辑日志复制（基于行）

另一种方法是，复制和存储引擎使用不同的日志格式，这样可以使复制日志从存储引擎内部分离出来。这种复制日志被称为逻辑日志，以将其与存储引擎的（物理）数据表示区分开来。

关系数据库的逻辑日志通常是以行的粒度描述对数据库表的写入的记录序列：

- 对于插入的行，日志包含所有列的新值。
- 对于删除的行，日志包含足够的信息来唯一标识已删除的行。通常是主键，但是如果表上没有主键，则需要记录所有列的旧值。
- 对于更新的行，日志包含足够的信息来唯一标识更新的行，以及所有列的新值（或至少所有已更改的列的新值）。

修改多行的事务会生成多个这样的日志记录，后面跟着一条记录，指出事务已经提交。MySQL的二进制日志（当配置为使用基于行的复制时）使用这种方法【17】。

由于逻辑日志与存储引擎内部分离，因此可以更容易地保持向后兼容，从而使领导者和跟随者能够运行不同版本的数据库软件甚至不同的存储引擎。

对于外部应用程序来说，逻辑日志格式也更容易解析。如果要将数据库的内容发送到外部系统（如数据），这一点很有用，例如复制到数据仓库进行离线分析，或建立自定义索引和缓存【18】。 这种技术被称为捕获数据变更（**change data capture**），第11章将重新讲到它。

基于触发器的复制

到目前为止描述的复制方法是由数据库系统实现的，不涉及任何应用程序代码。在很多情况下，这就是你想要的。但在某些情况下需要更多的灵活性。例如，如果您只想复制数据的一个子集，或者想从一种数据库复制到另一种数据库，或者如果您需要冲突解决逻辑（参阅“处理写入冲突”），则可能需要将复制移动到应用程序层。

一些工具，如Oracle Golden Gate 【19】，可以通过读取数据库日志，使得其他应用程序可以使用数据。另一种方法是使用许多关系数据库自带的功能：触发器和存储过程。

触发器允许您注册在数据库系统中发生数据更改（写入事务）时自动执行的自定义应用程序代码。触发器有机会将更改记录到一个单独的表中，使用外部程序读取这个表，再加上任何业务逻辑处理，会后将数据变更复制到另一个系统去。例如，Databus for Oracle 【20】和Bucardo for Postgres 【21】就是这样工作的。

基于触发器的复制通常比其他复制方法具有更高的开销，并且比数据库的内置复制更容易出错，也有很多限制。然而由于其灵活性，仍然是很有用的。

复制延迟问题

容忍节点故障只是需要复制的一个原因。正如在第二部分的介绍中提到的，另一个原因是可扩展性（处理比单个机器更多的请求）和延迟（让副本在地理位置上更接近用户）。

基于主库的复制要求所有写入都由单个节点处理，但只读查询可以由任何副本处理。所以对于读多写少的场景（Web上的常见模式），一个有吸引力的选择是创建很多从库，并将读请求分散到所有的从库上去。这样能减小主库的负载，并允许向最近的副本发送读请求。

在这种扩展体系结构中，只需添加更多的追随者，就可以提高只读请求的服务容量。但是，这种方法实际上只适用于异步复制——如果尝试同步复制到所有追随者，则单个节点故障或网络中断将使整个系统无法写入。而且越多的节点越有可能会被关闭，所以完全同步的配置是非常不可靠的。

不幸的是，当应用程序从异步从库读取时，如果从库落后，它可能会看到过时的信息。这会导致数据库中出现明显的不一致：同时对主库和从库执行相同的查询，可能得到不同的结果，因为并非所有的写入都反映在从库中。这种不一致只是一个暂时的状态——如果停止写入数据库并等待一段时间，从库最终会赶上并与主库保持一致。出于这个原因，这种效应被称为最终一致性（**eventually consistency**）[iii]【22,23】

> iii. 道格拉斯·特里（Douglas Terry）等人创造了术语最终一致性。【24】并经由Werner Vogels【22】推广，成为许多NoSQL项目的战吼。然而，不只有NoSQL数据库是最终一致的：关系型数据库中的异步复制追随者也有相同的特性。↩

“最终”一词故意含糊不清：总的来说，副本落后的程度是没有限制的。在正常的操作中，复制延迟（**replication lag**），即写入主库到反映至从库之间的延迟，可能仅仅是几分之一秒，在实践中并不显眼。但如果系统在接近极限的情况下运行，或网络中存在问题，延迟可以轻而易举地超过几秒，甚至几分钟。

因为滞后时间太长引入的不一致性，可不仅是一个理论问题，更是应用设计中会遇到的真实问题。本节将重点介绍三个由复制延迟问题的例子，并简述解决这些问题的一些方法。

读己之写

许多应用程序让用户提交一些数据，然后查看他们提交的内容。可能是用户数据库中的记录，也可能是对讨论主题的评论，或其他类似的内容。提交新数据时，必须将其发送给领导者，但是当用户查看数据时，可以从追随者读取。如果数据经常被查看，但只是偶尔写入，这是非常合适的。

但对于异步复制，问题就来了。如图5-3所示：如果用户在写入后马上就查看数据，则新数据可能尚未到达副本。对用户而言，看起来好像是刚提交的数据丢失了，用户会不高兴，可以理解。

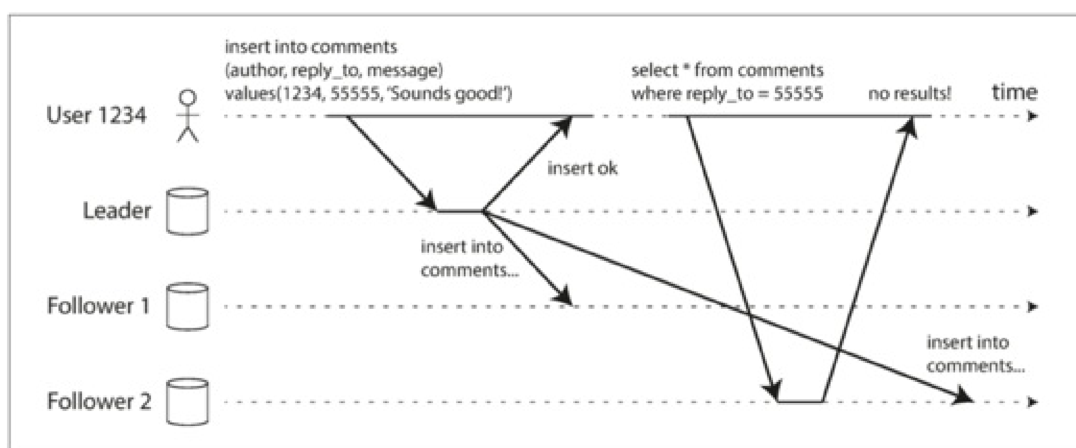


图**5-3** 用户写入后从旧副本中读取数据。需要写后读**(read-after-write)**的一致性来防止这种异常

在这种情况下，我们需要读写一致性（**read-after-write consistency**），也称为读己之写一致性（**read-your-writes consistency**）【24】。这是一个保证，如果用户重新加载页面，他们总会看到他们自己提交的任何更新。它不会对其他用户的写入做出承诺：其他用户的更新可能稍等才会看到。它保证用户自己的输入已被正确保存。

如何在基于领导者的复制系统中实现读后一致性？有各种可能的技术，这里说一些：

- 读用户可能已经修改过的内容时，都从主库读；这就要求有一些方法，不用实际查询就可以知道用户是否修改了某些东西。举个例子，社交网络上的用户个人资料信息通常只能由用户本人编辑，而不能由其他人编辑。因此一个简单的规则是：从主库读取用户自己的档案，在从库读取其他用户的档案。

- 如果应用中的大部分内容都可能被用户编辑，那这种方法就没用了，因为大部分内容都必须从主库读取（扩容读就没效果了）。在这种情况下可以使用其他标准来决定是否从主库读取。例如可以跟踪上次更新的时间，在上次更新后的一分钟内，从主库读。还可以监控从库的复制延迟，防止任向任何滞后超过一分钟到底从库发出查询。

- 客户端可以记住最近一次写入的时间戳，系统需要确保从库为该用户提供任何查询时，该时间戳前的变更都已经传播到了本从库中。如果当前从库不够新，则可以从另一个从库读，或者等待从库追赶上来。

  时间戳可以是逻辑时间戳（指示写入顺序的东西，例如日志序列号）或实际系统时钟（在这种情况下，时钟同步变得至关重要；参阅“不可靠的时钟”）。

- 如果您的副本分布在多个数据中心（出于可用性目的与用户尽量在地理上接近），则会增加复杂性。任何需要由领导者提供服务的请求都必须路由到包含主库的数据中心。

另一种复杂的情况是：如果同一个用户从多个设备请求服务，例如桌面浏览器和移动APP。这种情况下可能就需要提供跨设备的写后读一致性：如果用户在某个设备上输入了一些信息，然后在另一个设备上查看，则应该看到他们刚输入的信息。

在这种情况下，还有一些需要考虑的问题：

- 记住用户上次更新时间戳的方法变得更加困难，因为一台设备上运行的程序不知道另一台设备上发生了什么。元数据需要一个中心存储。
- 如果副本分布在不同的数据中心，很难保证来自不同设备的连接会路由到同一数据中心。（例如，用户的台式计算机使用家庭宽带连接，而移动设备使用蜂窝数据网络，则设备的网络路线可能完全不同）。如果你的方法需要读主库，可能首先需要把来自同一用户的请求路由到同一个数据中心。

单调读

从异步从库读取第二个异常例子是，用户可能会遇到时光倒流（**moving backward in time**）。

如果用户从不同从库进行多次读取，就可能发生这种情况。例如，图5-4显示了用户2345两次进行相同的查询，首先查询了一个延迟很小的从库，然后是一个延迟较大的从库。（如果用户刷新网页，而每个请求被路由到一个随机的服务器，这种情况是很有可能的。）第一个查询返回最近由用户1234添加的评论，但是第二个查询不返回任何东西，因为滞后的从库还没有拉取写入内容。在效果上相比第一个查询，第二个查询是在更早的时间点来观察系统。如果第一个查询没有返回任何内容，那问题并不大，因为用户2345可能不知道用户1234最近添加了评论。但如果用户2345先看见用户1234的评论，然后又看到它消失，那么对于用户2345，就很让人头大了。

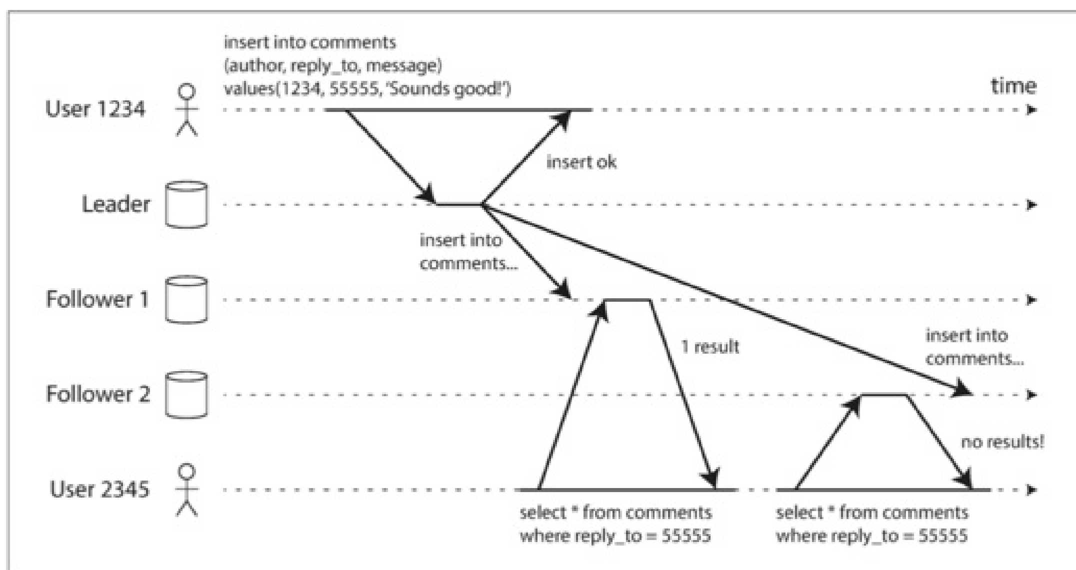


图**5-4** 用户首先从新副本读取，然后从旧副本读取。时光倒流。为了防止这种异常，我们需要单调的读取。

单调读（**Monotonic reads**）【23】是这种异常不会发生的保证。这是一个比强一致性（**strong consistency**）更弱，但比最终一致性（**eventually consistency**）更强的保证。当读取数据时，您可能会看到一个旧值；单调读取仅意味着如果一个用户顺序地进行多次读取，则他们不会看到时间后退，即，如果先前读取到较新的数据，后续读取不会得到更旧的数据。

实现单调读取的一种方式是确保每个用户总是从同一个副本进行读取（不同的用户可以从不同的副本读取）。例如，可以基于用户ID的散列来选择副本，而不是随机选择副本。但是，如果该副本失败，用户的查询将需要重新路由到另一个副本。

一致前缀读

第三个复制延迟例子违反了因果律。 想象一下Poons先生和Cake夫人之间的以下简短对话：

> *Mr. Poons* Mrs. Cake，你能看到多远的未来？
>
> *Mrs. Cake* 通常约十秒钟，Mr. Poons.

这两句话之间有因果关系：Cake夫人听到了Poons先生的问题并回答了这个问题。

现在，想象第三个人正在通过从库来听这个对话。 Cake夫人说的内容是从一个延迟很低的从库读取的，但Poons先生所说的内容，从库的延迟要大的多（见图5-5）。 于是，这个观察者会听到以下内容：

> *Mrs. Cake* 通常约十秒钟，Mr. Poons.
>
> *Mr. Poons* Mrs. Cake，你能看到多远的未来？

对于观察者来说，看起来好像Cake夫人在Poons先生发问前就回答了这个问题。 这种超能力让人印象深刻，但也会把人搞糊涂。【25】。

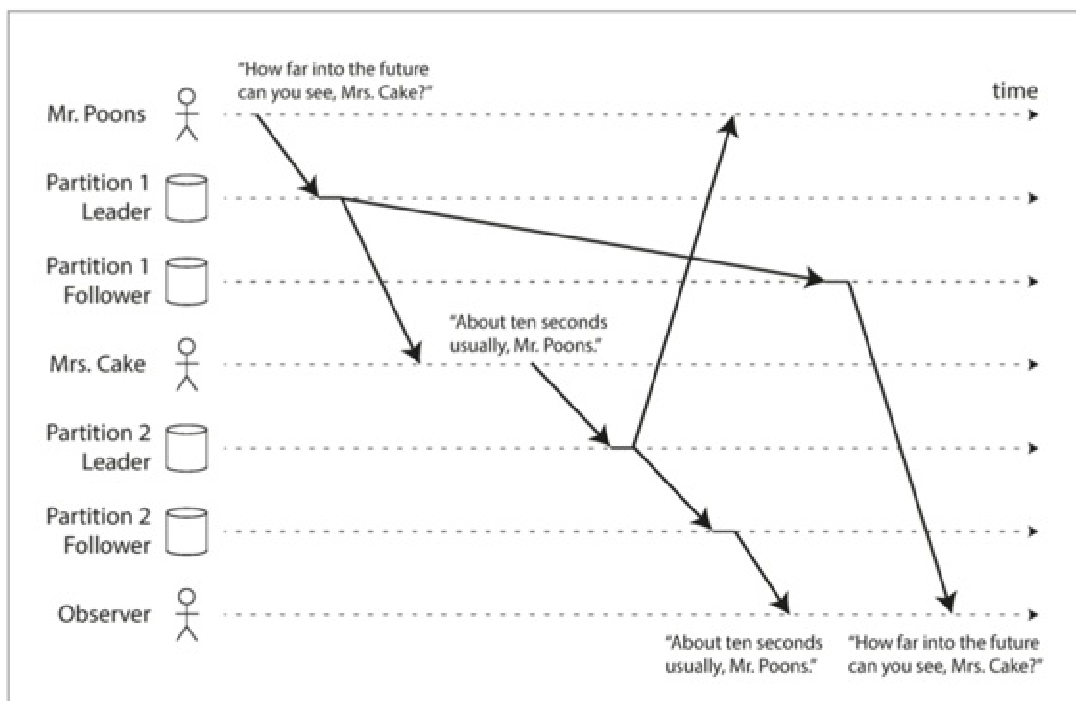


图**5-5** 如果某些分区的复制速度慢于其他分区，那么观察者在看到问题之前可能会看到答案。

防止这种异常，需要另一种类型的保证：一致前缀读（**consistent prefix reads**）【23】。 这个保证说：如果一系列写入按某个顺序发生，那么任何人读取这些写入时，也会看见它们以同样的顺序出现。

这是分区（**partitioned**）（分片（**sharded**））数据库中的一个特殊问题，将在第6章中讨论。如果数据库总是以相同的顺序应用写入，则读取总是会看到一致的前缀，所以这种异常不会发生。但是在许多分布式数据库中，不同的分区独立运行，因此不存在全局写入顺序：当用户从数据库中读取数据时，可能会看到数据库的某些部分处于较旧的状态，而某些处于较新的状态。

一种解决方案是，确保任何因果相关的写入都写入相同的分区。对于某些无法高效完成这种操作的应用，还有一些显式跟踪因果依赖关系的算法，本书将在“关系与并发”一节中返回这个主题。

复制延迟的解决方案

在使用最终一致的系统时，如果复制延迟增加到几分钟甚至几小时，则应该考虑应用程序的行为。如果答案是“没问题”，那很好。但如果结果对于用户来说是不好体验，那么设计系统来提供更强的保证是很重要的，例如写后读。明明是异步复制却假设复制是同步的，这是很多麻烦的根源。

如前所述，应用程序可以提供比底层数据库更强有力的保证，例如通过主库进行某种读取。但在应用程序代码中处理这些问题是复杂的，容易出错。

如果应用程序开发人员不必担心微妙的复制问题，并可以信赖他们的数据库“做了正确的事情”，那该多好呀。这就是事务（**transaction**）存在的原因：数据库通过事务提供强大的保证，所以应用程序可以更假简单。

单节点事务已经存在了很长时间。然而在走向分布式（复制和分区）数据库时，许多系统放弃了事务。声称事务在性能和可用性上的代价太高，并断言在可扩展系统中最终一致性是不可避免的。这个叙述有一些道理，但过于简单了，本书其余部分将提出更为细致的观点。第七章和第九章将回到事务的话题，并讨论一些替代机制。

多主复制

本章到目前为止，我们只考虑使用单个领导者的复制架构。 虽然这是一种常见的方法，但也有一些有趣的选择。

基于领导者的复制有一个主要的缺点：只有一个主库，而所有的写入都必须通过它。如果出于任何原因（例如和主库之间的网络连接中断）无法连接到主库， 就无法向数据库写入。

> iv. 如果数据库被分区（见第6章），每个分区都有一个领导。 不同的分区可能在不同的节点上有其领导者，但是每个分区必须有一个领导者节点。 ↩

基于领导者的复制模型的自然延伸是允许多个节点接受写入。 复制仍然以同样的方式发生：处理写入的每个节点都必须将该数据更改转发给所有其他节点。 称之为多领导者配置（也称多主、多活复制）。 在这种情况下，每个领导者同时扮演其他领导者的追随者。

多主复制的应用场景

在单个数据中心内部使用多个主库很少是有意义的，因为好处很少超过复杂性的代价。 但在一些情况下，多活配置是也合理的。

运维多个数据中心

假如你有一个数据库，副本分散在好几个不同的数据中心（也许这样可以容忍单个数据中心的故障，或地理上更接近用户）。 使用常规的基于领导者的复制设置，主库必须位于其中一个数据中心，且所有写入都必须经过该数据中心。

多领导者配置中可以在每个数据中心都有主库。 图5-6展示了这个架构的样子。 在每个数据中心内使用常规的主从复制;；在数据中心之间，每个数据中心的主库都会将其更改复制到其他数据中心的主库中。

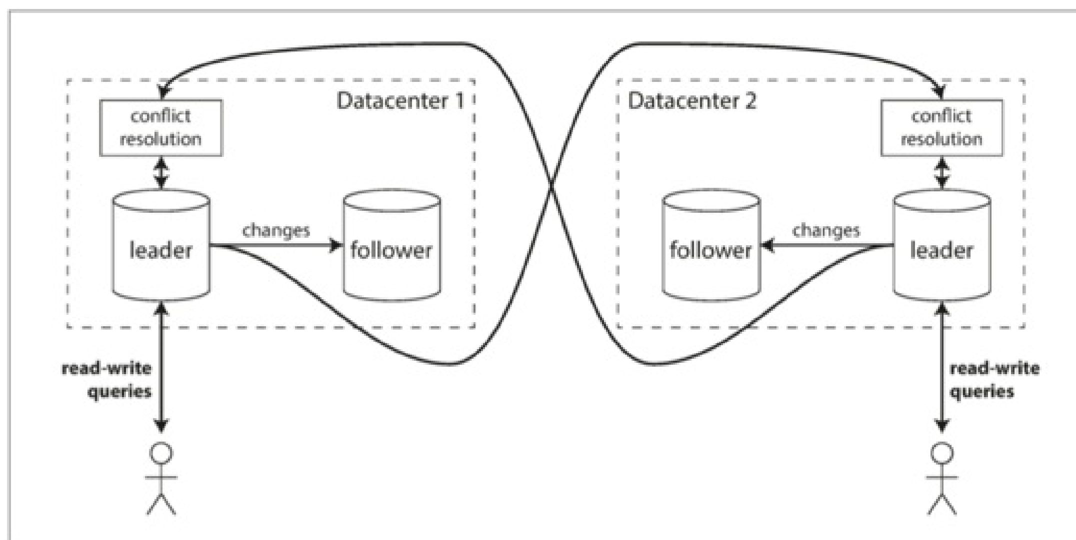


图**5-6** 跨多个数据中心的多主复制

我们来比较一下在运维多个数据中心时，单主和多主的适应情况。

性能

在单活配置中，每个写入都必须穿过互联网，进入主库所在的数据中心。这可能会增加写入时间，并可能违背了设置多个数据中心的初心。在多活配置中，每个写操作都可以在本地数据中心进行处理，并与其他数据中心异步复制。因此，数据中心之间的网络延迟对用户来说是透明的，这意味着感觉到的性能可能会更好。

容忍数据中心停机

在单主配置中，如果主库所在的数据中心发生故障，故障转移可以使另一个数据中心里的追随者成为领导者。在多活配置中，每个数据中心可以独立于其他数据中心继续运行，并且当发生故障的数据中心归队时，复制会自动赶上。

容忍网络问题

数据中心之间的通信通常穿过公共互联网，这可能不如数据中心内的本地网络可靠。单主配置对这数据中心间的连接问题非常敏感，因为通过这个连接进行的写操作是同步的。采用异步复制功能的多活配置通常能更好地承受网络问题：临时的网络中断并不会妨碍正在处理的写入。

有些数据库默认情况下支持多主配置，但使用外部工具实现也很常见，例如用于MySQL的Tungsten Replicator 【26】，用于PostgreSQL的BDR【27】以及用于Oracle的GoldenGate【19】。

尽管多主复制有这些优势，但也有一个很大的缺点：两个不同的数据中心可能会同时修改相同的数据，写冲突是必须解决的（如图5-6中“冲突解决”）。本书将在“处理写入冲突”中详细讨论这个问题。

由于多主复制在许多数据库中都属于改装的功能，所以常常存在微妙的配置缺陷，且经常与其他数据库功能之间出现意外的反应。例如自增主键、触发器、完整性约束等，都可能会有麻烦。因此，多主复制往往被认为是危险的领域，应尽可能避免【28】。

需要离线操作的客户端

多主复制的另一种适用场景是：应用程序在断网之后仍然需要继续工作。

例如，考虑手机，笔记本电脑和其他设备上的日历应用。无论设备目前是否有互联网连接，你需要能随时查看你的会议（发出读取请求），输入新的会议（发出写入请求）。如果在离线状态下进行任何更改，则设备下次上线时，需要与服务器和其他设备同步。

在这种情况下，每个设备都有一个充当领导者的本地数据库（它接受写请求），并且在所有设备上的日历副本之间同步时，存在异步的多主复制过程。复制延迟可能是几小时甚至几天，具体取决于何时可以访问互联网。

从架构的角度来看，这种设置实际上与数据中心之间的多领导者复制类似，每个设备都是一个“数据中心”，而它们之间的网络连接是极度不可靠的。从历史上各类日历同步功能的破烂实现可以看出，想把多活配好是多么困难的一件事。

有一些工具旨在使这种多领导者配置更容易。例如，CouchDB就是为这种操作模式而设计的【29】。

协同编辑

实时协作编辑应用程序允许多个人同时编辑文档。例如，Etherpad 【30】和Google Docs【31】允许多人同时编辑文本文档或电子表格（该算法在“自动冲突解决”中简要讨论）。我们通常不会将协作式编辑视为数据库复制问题，但与前面提到的离线编辑用例有许多相似之处。当一个用户编辑文档时，所做的更改将立即应用到其本地副本（Web浏览器或客户端应用程序中的文档状态），并异步复制到服务器和编辑同一文档的任何其他用户。

如果要保证不会发生编辑冲突，则应用程序必须先取得文档的锁定，然后用户才能对其进行编辑。如果另一个用户想要编辑同一个文档，他们首先必须等到第一个用户提交修改并释放锁定。这种协作模式相当于在领导者上进行交易的单领导者复制。

但是，为了加速协作，您可能希望将更改的单位设置得非常小（例如，一个按键），并避免锁定。这种方法允许多个用户同时进行编辑，但同时也带来了多领导者复制的所有挑战，包括需要解决冲突【32】。

处理写入冲突

多领导者复制的最大问题是可能发生写冲突，这意味着需要解决冲突。

例如，考虑一个由两个用户同时编辑的维基页面，如图5-7所示。用户1将页面的标题从A更改为B，并且用户2同时将标题从A更改为C。每个用户的更改已成功应用到其本地主库。但当异步复制时，会发现冲突【33】。单主数据库中不会出现此问题。

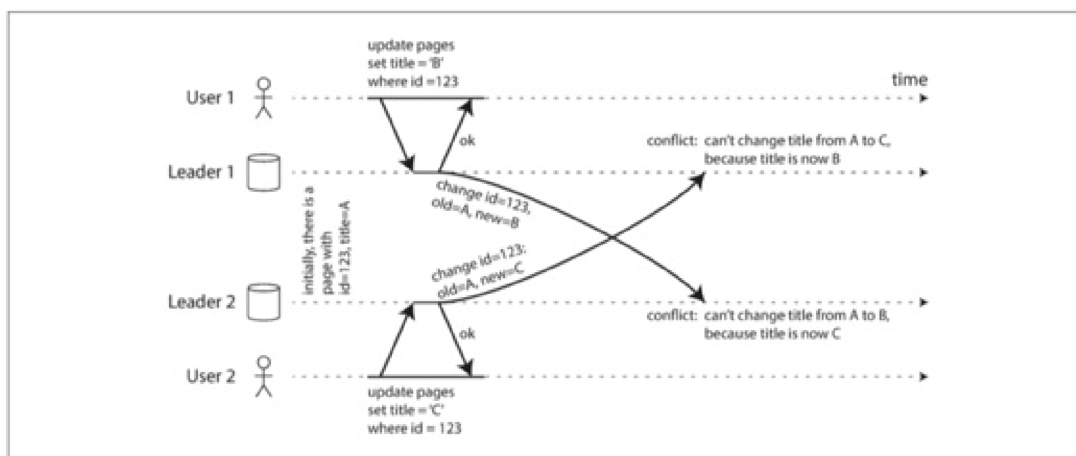


图**5-7** 两个主库同时更新同一记录引起的写入冲突

同步与异步冲突检测

在单主数据库中，第二个写入将被阻塞，并等待第一个写入完成，或中止第二个写入事务，强制用户重试。另一方面，在多活配置中，两个写入都是成功的，并且在稍后的时间点仅仅异步地检测到冲突。那时要求用户解决冲突可能为时已晚。

原则上，可以使冲突检测同步 - 即等待写入被复制到所有副本，然后再告诉用户写入成功。但是，通过这样做，您将失去多主复制的主要优点：允许每个副本独立接受写入。如果您想要同步冲突检测，那么您可以使用单主程序复制。

避免冲突

处理冲突的最简单的策略就是避免它们：如果应用程序可以确保特定记录的所有写入都通过同一个领导者，那么冲突就不会发生。由于多领导者复制处理的许多实现冲突相当不好，避免冲突是一个经常推荐的方法【34】。

例如，在用户可以编辑自己的数据的应用程序中，可以确保来自特定用户的请求始终路由到同一数据中心，并使用该数据中心的领导者进行读写。不同的用户可能有不同的“家庭”数据中心（可能根据用户的地理位置选择），但从任何用户的角度来看，配置基本上都是单一的领导者。

但是，有时您可能需要更改指定的记录的主库——可能是因为一个数据中心出现故障，您需要将流量重新路由到另一个数据中心，或者可能是因为用户已经迁移到另一个位置，现在更接近不同的数据中心。在这种情况下，冲突避免会中断，你必须处理不同主库同时写入的可能性。

收敛至一致的状态

单主数据库按顺序应用写操作：如果同一个字段有多个更新，则最后一个写操作将确定该字段的最终值。

在多主配置中，写入顺序没有定义，所以最终值应该是什么并不清楚。在图5-7中，在主库1中标题首先更新为B而后更新为C；在主库2中，首先更新为C，然后更新为B。两个顺序都不是“更正确”的。

如果每个副本只是按照它看到写入的顺序写入，那么数据库最终将处于不一致的状态：最终值将是在主库1的C和主库2的B。这是不可接受的，每个复制方案都必须确保数据在所有副本中最终都是相同的。因此，数据库必须以一种收敛（**convergent**）的方式解决冲突，这意味着所有副本必须在所有变更复制完成时收敛至一个相同的最终值。

实现冲突合并解决有多种途径：

- 给每个写入一个唯一的ID（例如，一个时间戳，一个长的随机数，一个UUID或者一个键和值的哈希），挑选最高ID的写入作为胜利者，并丢弃其他写入。如果使用时间戳，这种技术被称为最后写入胜利（**LWW, last write wins**）。虽然这种方法很流行，但是很容易造成数据丢失【35】。我们将在本章末尾更详细地讨论LWW。
- 为每个副本分配一个唯一的ID，ID编号更高的写入具有更高的优先级。这种方法也意味着数据丢失。
- 以某种方式将这些值合并在一起 - 例如，按字母顺序排序，然后连接它们（在图5-7中，合并的标题可能类似于“B/C”）。
- 在保留所有信息的显式数据结构中记录冲突，并编写解决冲突的应用程序代码（也许通过提示用户的方式）。

自定义冲突解决逻辑

作为解决冲突最合适的方法可能取决于应用程序，大多数多主复制工具允许使用应用程序代码编写冲突解决逻辑。该代码可以在写入或读取时执行：

写时执行

只要数据库系统检测到复制更改日志中存在冲突，就会调用冲突处理程序。例如，Bucardo允许您为此编写一段Perl代码。这个处理程序通常不能提示用户——它在后台进程中运行，并且必须快速执行。

读时执行

当检测到冲突时，所有冲突写入被存储。下一次读取数据时，会将这些多个版本的数据返回给应用程序。应用程序可能会提示用户或自动解决冲突，并将结果写回数据库。例如，CouchDB以这种方式工作。

请注意，冲突解决通常适用于单个行或文档层面，而不是整个事务【36】。因此，如果您有一个事务会原子性地进行几次不同的写入（请参阅第7章），则对于冲突解决而言，每个写入仍需分开单独考虑。

> 题外话：自动冲突解决
>
> 冲突解决规则可能很快变得复杂，并且自定义代码可能容易出错。亚马逊是一个经常被引用的例子，由于冲突解决处理程序令人惊讶的效果：一段时间以来，购物车上的冲突解决逻辑将保留添加到购物车的物品，但不包括从购物车中移除的物品。因此，顾客有时会看到物品重新出现在他们的购物车中，即使他们之前已经被移走【37】。
>
> 已经有一些有趣的研究来自动解决由于数据修改引起的冲突。有几行研究值得一提：
>
> - 无冲突复制数据类型（**Conflict-free replicated datatypes**）（CRDT）【32,38】是可以由多个用户同时编辑的集合，映射，有序列表，计数器等的一系列数据结构，它们以合理的方式自动解决冲突。一些CRDT已经在Riak 2.0中实现【39,40】。
> - 可合并的持久数据结构（**Mergeable persistent data structures**）【41】显式跟踪历史记录，类似于Git版本控制系统，并使用三向合并功能（而CRDT使用双向合并）。
> - 可执行的转换（**operational transformation**）[42]是Etherpad 【30】和Google Docs 【31】等合作编辑应用背后的冲突解决算法。它是专为同时编辑项目的有序列表而设计的，例如构成文本文档的字符列表。
>
> 这些算法在数据库中的实现还很年轻，但很可能将来它们将被集成到更多的复制数据系统中。自动冲突解决方案可以使应用程序处理多领导者数据同步更为简单。

什么是冲突？

有些冲突是显而易见的。在图5-7的例子中，两个写操作并发地修改了同一条记录中的同一个字段，并将其设置为两个不同的值。毫无疑问这是一个冲突。

其他类型的冲突可能更为微妙，难以发现。例如，考虑一个会议室预订系统：它记录谁腚了哪个时间段的哪个房间。应用需要确保每个房间只有一组人同时预定（即不得有相同房间的重叠预订）。在这种情况下，如果同时为同一个房间创建两个不同的预订，则可能会发生冲突。即使应用程序在允许用户进行预订之前检查可用性，如果两次预订是由两个不同的领导者进行的，则可能会有冲突。

现在还没有一个现成的答案，但在接下来的章节中，我们将追溯到对这个问题有很好的理解。我们将在第7章中看到更多的冲突示例，在第12章中我们将讨论用于检测和解决复制系统中冲突的可扩展方法。

多主复制拓扑

复制拓扑描述写入从一个节点传播到另一个节点的通信路径。如果你有两个领导者，如图5-7所示，只有一个合理的拓扑结构：领导者1必须把他所有的写到领导者2，反之亦然。有两个以上的领导，各种不同的拓扑是可能的。图5-8举例说明了一些例子。

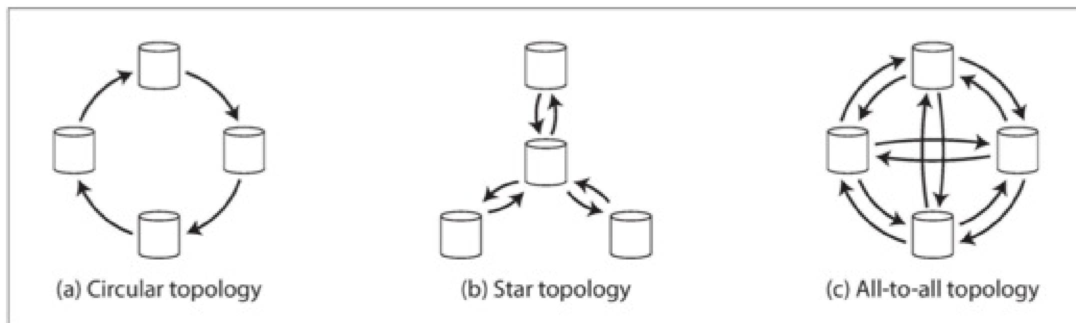


图**5-8** 三个可以设置多领导者复制的示例拓扑。

最普遍的拓扑是全部到全部（图5-8 [c]），其中每个领导者将其写入每个其他领导。但是，也会使用更多受限制的拓扑：例如，默认情况下，MySQL仅支持环形拓扑（**circular topology**）【34】，其中每个节点接收来自一个节点的写入，并将这些写入（加上自己的任何写入）转发给另一个节点。另一种流行的拓扑结构具有星形的形状[V]。个指定的根节点将写入转发给所有其他节点。星型拓扑可以推广到树。

> V. 不要与星型模式混淆（请参阅“分析模式：星型还是雪花”），其中描述了数据模型的结构，而不是节点之间的通信拓扑。↩

在圆形和星形拓扑中，写入可能需要在到达所有副本之前通过多个节点。因此，节点需要转发从其他节点收到的数据更改。为了防止无限复制循环，每个节点被赋予一个唯一的标识符，并且在复制日志中，每个写入都被标记了所有已经通过的节点的标识符【43】。当一个节点收到用自己的标识符标记的数据更改时，该数据更改将被忽略，因为节点知道它已经被处理。

循环和星型拓扑的问题是，如果只有一个节点发生故障，则可能会中断其他节点之间的复制消息流，导致它们无法通信，直到节点修复。拓扑结构可以重新配置为在发生故障的节点上工作，但在大多数部署中，这种重新配置必须手动完成。更密集连接的拓扑结构（例如全部到全部）的容错性更好，因为它允许消息沿着不同的路径传播，避免单点故障。

另一方面，全能拓扑也可能有问题。特别是，一些网络链接可能比其他网络链接更快（例如，由于网络拥塞），结果是一些复制消息可能“超过”其他复制消息，如图5-9所示。

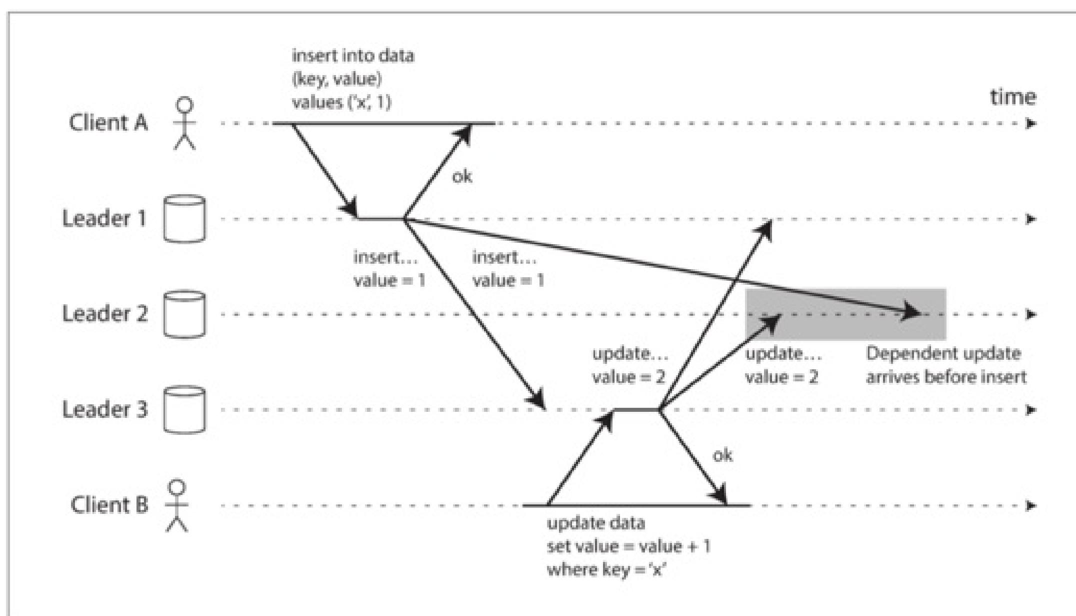


图**5-9** 使用多主程序复制时，可能会在某些副本中写入错误的顺序。

在图5-9中，客户端A向林登万（**Leader One**）的表中插入一行，客户端B在主库3上更新该行。然而，主库2可以以不同的顺序接收写入：它可以首先接收更新（其中，从它的角度来看，是对数据库中不存在的行的更新），并且仅在稍后接收到相应的插入（其应该在更新之前）。

这是一个因果关系的问题，类似于我们在"一致前缀读"中看到的：更新取决于先前的插入，所以我们需要确保所有节点先处理插入，然后再处理更新。仅仅在每一次写入时添加一个时间戳是不够的，因为时钟不可能被充分地同步，以便在主库2处正确地排序这些事件（见第8章）。

要正确排序这些事件，可以使用一种称为版本向量（**version vectors**）的技术，本章稍后将讨论这种技术（参阅"检测并发写入"）。然而，冲突检测技术在许多多领导者复制系统中执行得不好。例如，在撰写本文时，PostgreSQL BDR不提供写入的因果排序【27】，而Tungsten Replicator for MySQL甚至不尝试检测冲突【34】。

如果您正在使用具有多领导者复制功能的系统，那么应该了解这些问题，仔细阅读文档，并彻底测试您的数据库，以确保它确实提供了您认为具有的保证。

无主复制

我们在本章到目前为止所讨论的复制方法 ——单主复制、多主复制——都是这样的想法：客户端向一个主库发送写请求，而数据库系统负责将写入复制到其他副本。主库决定写入的顺序，而从库按相同顺序应用主库的写入。

一些数据存储系统采用不同的方法，放弃主库的概念，并允许任何副本直接接受来自客户端的写入。最早的一些的复制数据系统是无领导的（**leaderless**）【1,44】，但是在关系数据库主导的时代，这个想法几乎已被忘却。在亚马逊将其用于其内部的Dynamo系统[vi]之后，它再

一次成为数据库的一种时尚架构【37】。（Dynamo不适用于Amazon以外的用户。 令人困惑的是，AWS提供了一个名为DynamoDB的托管数据库产品，它使用了完全不同的体系结构：它基于单主程序复制。） Riak，Cassandra和Voldemort是由Dynamo启发的无领导复制模型的开源数据存储，所以这类数据库也被称为*Dynamo*风格。

> vi. Dynamo不适用于Amazon以外的用户。 令人困惑的是，AWS提供了一个名为DynamoDB的托管数据库产品，它使用了完全不同的体系结构：它基于单引导程序复制。↩

在一些无领导者的实现中，客户端直接将写入发送到到几个副本中，而另一些情况下，一个协调者（**coordinator**）节点代表客户端进行写入。但与主库数据库不同，协调员不执行特定的写入顺序。我们将会看到，这种设计上的差异对数据库的使用方式有着深远的影响。

当节点故障时写入数据库

假设你有一个带有三个副本的数据库，而其中一个副本目前不可用，或许正在重新启动以安装系统更新。在基于主机的配置中，如果要继续处理写入，则可能需要执行故障切换（参阅「处理节点宕机」）。

另一方面，在无领导配置中，故障切换不存在。图5-10显示了发生了什么事情：客户端（用户1234）并行发送写入到所有三个副本，并且两个可用副本接受写入，但是不可用副本错过了它。假设三个副本中的两个承认写入是足够的：在用户1234已经收到两个确定的响应之后，我们认为写入成功。客户简单地忽略了其中一个副本错过了写入的事实。

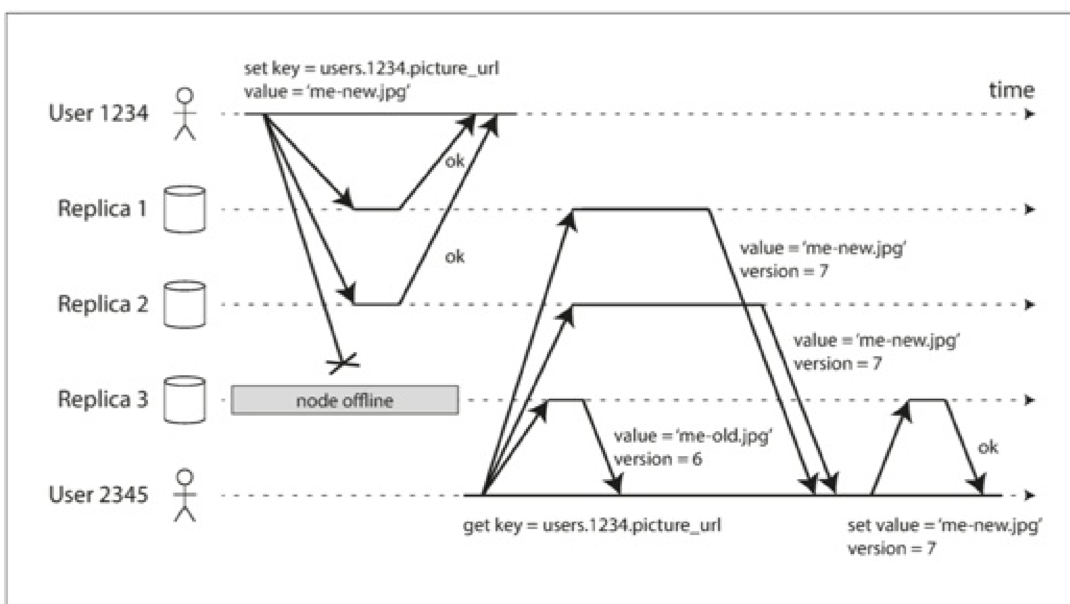


图**5-10** 仲裁写入，法定读取，并在节点中断后读取修复。

现在想象一下，不可用的节点重新联机，客户端开始读取它。节点关闭时发生的任何写入都从该节点丢失。因此，如果您从该节点读取数据，则可能会将陈旧（过时）值视为响应。

为了解决这个问题，当一个客户端从数据库中读取数据时，它不仅仅发送它的请求到一个副本：读请求也被并行地发送到多个节点。客户可能会从不同的节点获得不同的响应。即来自一个节点的最新值和来自另一个节点的陈旧值。版本号用于确定哪个值更新（参阅“检测并发写入”）。

读修复和反熵

复制方案应确保最终将所有数据复制到每个副本。在一个不可用的节点重新联机之后，它如何赶上它错过的写入？

在Dynamo风格的数据存储中经常使用两种机制：

读修复（***Read repair***）

当客户端并行读取多个节点时，它可以检测到任何陈旧的响应。例如，在图5-10中，用户2345获得了来自Replica 3的版本6值和来自副本1和2的版本7值。客户端发现副本3具有陈旧值，并将新值写回复制品。这种方法适用于频繁阅读的值。

反熵过程（***Anti-entropy process***）

此外，一些数据存储具有后台进程，该进程不断查找副本之间的数据差异，并将任何缺少的数据从一个副本复制到另一个副本。与基于领导者的复制中的复制日志不同，此反熵过程不会以任何特定的顺序复制写入，并且在复制数据之前可能会有显着的延迟。

并不是所有的系统都实现了这两个;例如，Voldemort目前没有反熵过程。请注意，如果没有反熵过程，某些副本中很少读取的值可能会丢失，从而降低了持久性，因为只有在应用程序读取值时才执行读取修复。

读写的法定人数

在图5-10的示例中，我们认为即使仅在三个副本中的两个上进行处理，写入仍然是成功的。如果三个副本中只有一个接受了写入，会怎样？我们能推多远呢？

如果我们知道，每个成功的写操作意味着在三个副本中至少有两个出现，这意味着至多有一个副本可能是陈旧的。因此，如果我们从至少两个副本读取，我们可以确定至少有一个是最新的。如果第三个副本停机或响应速度缓慢，则读取仍可以继续返回最新值。

更一般地说，如果有n个副本，每个写入必须由w节点确认才能被认为是成功的，并且我们必须至少为每个读取查询r个节点。（在我们的例子中，$n = 3，w = 2，r = 2$）。只要$w + r> n$，我们期望在读取时获得最新的值，因为r个读取中至少有一个节点是最新的。遵循这些r值，w值的读写称为法定人数（quorum）[vii]的读和写。【44】，你可以认为，r和w是有效读写所需的最低票数。

> [vii]. 有时候这种法定人数被称为严格的法定人数，相对“松散的法定人数”而言（见“松散法定人数与带提示的接力”）↩

在Dynamo风格的数据库中，参数n，w和r通常是可配置的。一个常见的选择是使n为奇数（通常为3或5）并设置 $w = r =（n + 1）/ 2$（向上取整）。但是可以根据需要更改数字。例如，设置$w = n$和$r = 1$的写入很少且读取次数较多的工作负载可能会受益。这使得读取速度更快，但具有只有一个失败节点导致所有数据库写入失败的缺点。

> 集群中可能有多于n的节点。（集群的机器数可能多于副本书目），但是任何给定的值只能存储在n个节点上。 这允许对数据集进行分区，从而支持可以放在一个节点上的数据集更大的数据集。 将在第6章回到分区。

仲裁条件$w + r> n$允许系统容忍不可用的节点，如下所示：

- 如果$w <n$，如果节点不可用，我们仍然可以处理写入。
- 如果$r <n$，如果节点不可用，我们仍然可以处理读取。
- 对于$n = 3，w = 2，r = 2$，我们可以容忍一个不可用的节点。
- 对于$n = 5，w = 3，r = 3$，我们可以容忍两个不可用的节点。 这个案例如图5-11所示。
- 通常，读取和写入操作始终并行发送到所有n个副本。 参数w和r决定我们等待多少个节点，即在我们认为读或写成功之前，有多少个节点需要报告成功。

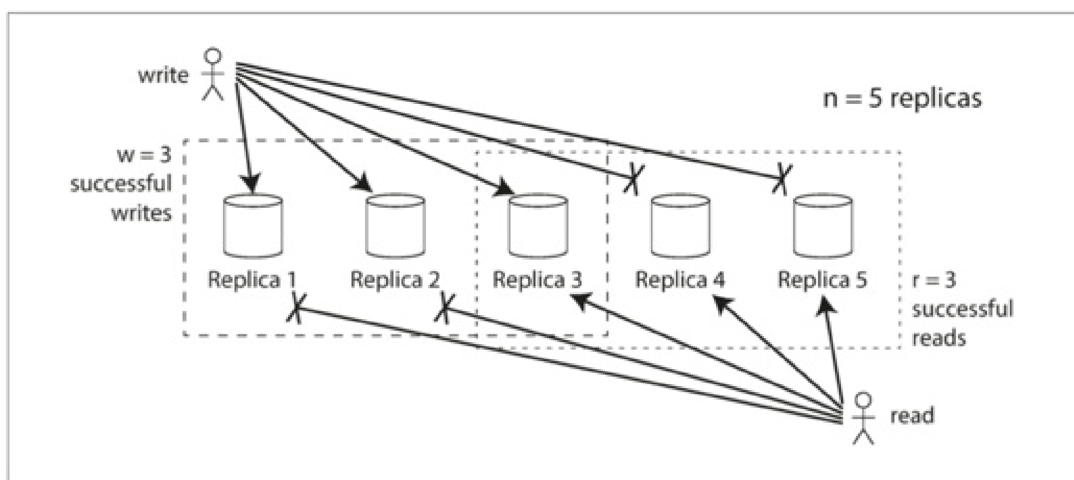


图**5-11** 如果**$w + r > n$**，读取**r**个副本，至少有一个**r**副本必然包含了最近的成功写入

如果少于所需的w或r节点可用，则写入或读取将返回错误。 由于许多原因，节点可能不可用：因为由于执行操作的错误（由于磁盘已满而无法写入）导致节点关闭（崩溃，关闭电源），由于客户端和服务器之间的网络中断 节点，或任何其他原因。 我们只关心节点是否返回了成功的响应，而不需要区分不同类型的错误。

仲裁一致性的局限性

如果你有n个副本，并且你选择w和r，使得$w + r> n$，你通常可以期望每个读取返回为一个键写的最近的值。情况就是这样，因为你写的节点集合和你读过的节点集合必须重叠。也就是说，您读取的节点中必须至少有一个具有最新值的节点（如图5-11所示）。

通常，r和w被选为多数（超过 $n/2$ ）节点，因为这确保了$w + r > n$，同时仍然容忍多达$n/2$个节点故障。但是，法定人数不一定必须是大多数，只是读写使用的节点交集至少需要包括一个节点。其他法定人数的配置是可能的，这使得分布式算法的设计有一定的灵活性【45】。

您也可以将w和r设置为较小的数字，以使$w + r \leq n$（即法定条件不满足）。在这种情况下，读取和写入操作仍将被发送到n个节点，但操作成功只需要少量的成功响应。

较小的w和r更有可能会读取过时的数据，因为您的读取更有可能不包含具有最新值的节点。另一方面，这种配置允许更低的延迟和更高的可用性：如果存在网络中断，并且许多副本变得无法访问，则可以继续处理读取和写入的机会更大。只有当可达副本的数量低于w或r时，数据库才分别变得不可用于写入或读取。

但是，即使在$w + r > n$的情况下，也可能存在返回陈旧值的边缘情况。这取决于实现，但可能的情况包括：

- 如果使用松散的法定人数（见“松散法定人数与带提示的接力”），w个写入和r个读取落在完全不同的节点上，因此r节点和w之间不再保证有重叠节点【46】。
- 如果两个写入同时发生，不清楚哪一个先发生。在这种情况下，唯一安全的解决方案是合并并发写入（请参阅第171页的“处理写入冲突”）。如果根据时间戳（最后写入成功）挑选出胜者，则由于时钟偏差[35]，写入可能会丢失。我们将返回“检测并发写入”中的此主题。
- 如果写操作与读操作同时发生，写操作可能仅反映在某些副本上。在这种情况下，不确定读取是返回旧值还是新值。
- 如果写操作在某些副本上成功，而在其他节点上失败（例如，因为某些节点上的磁盘已满），在小于w个副本上写入成功。所以整体判定写入失败，但整体写入失败并没有在写入成功的副本上回滚。这意味着如果一个写入虽然报告失败，后续的读取仍然可能会读取这次失败写入的值【47】。
- 如果携带新值的节点失败，需要读取其他带有旧值的副本。并且其数据从带有旧值的副本中恢复，则存储新值的副本数可能会低于w，从而打破法定人数条件。
- 即使一切工作正常，有时也会不幸地出现关于时序（**timing**）的边缘情况，在这种情况下，您可能会感到不安，因为我们将在第334页上的“线性化和法定人数”中看到。

因此，尽管法定人数似乎保证读取返回最新的写入值，但在实践中并不那么简单。 Dynamo风格的数据库通常针对可以忍受最终一致性的用例进行优化。允许通过参数w和r来调整读取陈旧值的概率，但把它们当成绝对的保证是不明智的。

尤其是，通常没有得到“与延迟有关的问题”（读取您的写入，单调读取或一致的前缀读取）中讨论的保证，因此前面提到的异常可能会发生在应用程序中。更强有力的保证通常需要事务或共识。我们将在第七章和第九章回到这些话题。

监控陈旧度

从运维的角度来看，监视你的数据库是否返回最新的结果是很重要的。即使应用可以容忍陈旧的读取，您也需要了解复制的健康状况。如果显着落后，应该提醒您，以便您可以调查原因（例如，网络中的问题或超载节点）。

对于基于领导者的复制，数据库通常会公开复制滞后的度量标准，您可以将其提供给监视系统。这是可能的，因为写入按照相同的顺序应用于领导者和追随者，并且每个节点在复制日志中具有一个位置（在本地应用的写入次数）。通过从领导者的当前位置中减去随从者的当前位置，您可以测量复制滞后量。

然而，在无领导者复制的系统中，没有固定的写入顺序，这使得监控变得更加困难。而且，如果数据库只使用读取修复（没有反熵过程），那么对于一个值可能会有多大的限制是没有限制的 - 如果一个值很少被读取，那么由一个陈旧副本返回的值可能是古老的。

已经有一些关于衡量无主复制数据库中的复制陈旧度的研究，并根据参数n，w和r来预测陈旧读取的预期百分比【48】。不幸的是，这还不是很常见的做法，但是将过时测量值包含在数据库的标准度量标准中是一件好事。最终的一致性是故意模糊的保证，但是对于可操作性来说，能够量化“最终”是很重要的。

松散法定人数与带提示的接力

合理配置的法定人数可以使数据库无需故障切换即可容忍个别节点的故障。也可以容忍个别节点变慢，因为请求不必等待所有n个节点响应——当w或r节点响应时它们可以返回。对于需要高可用、低延时、且能够容忍偶尔读到陈旧值的应用场景来说，这些特性使无主复制的数据库很有吸引力。

然而，法定人数（如迄今为止所描述的）并不像它们可能的那样具有容错性。网络中断可以很容易地将客户端从大量的数据库节点上切断。虽然这些节点是活着的，而其他客户端可能能够连接到它们，但是从数据库节点切断的客户端，它们也可能已经死亡。在这种情况下，剩余的可用节点可能会少于可用节点，因此客户端可能无法达到法定人数。

在一个大型的群集中（节点数量明显多于n个），网络中断期间客户端可能连接到某些数据库节点，而不是为了为特定值组成法定人数的节点们。在这种情况下，数据库设计人员需要权衡一下：

- 将错误返回给我们无法达到w或r节点的法定数量的所有请求是否更好？
- 或者我们是否应该接受写入，然后将它们写入一些可达的节点，但不在n值通常存在的n个节点之间？

后者被认为是一个松散的法定人数（**sloppy quorum**）【37】：写和读仍然需要w和r成功的响应，但是那些可能包括不在指定的n个“主”节点中的值。比方说，如果你把自己锁在房子外面，你可能会敲开邻居的门，问你是否可以暂时停留在沙发上。

一旦网络中断得到解决，代表另一个节点临时接受的一个节点的任何写入都被发送到适当的“本地”节点。这就是所谓的带提示的接力（**hinted handoff**）。（一旦你再次找到你的房子的钥匙，你的邻居礼貌地要求你离开沙发回家。）

松散法定人数提高写入可用性特别有用：只要有任何w节点可用，数据库就可以接受写入。然而，这意味着即使当$w + r> n$时，也不能确定读取某个键的最新值，因为最新的值可能已经临时写入了n之外的某些节点【47】。

因此，在传统意义上，一个松散的法定人数实际上不是一个法定人数。这只是一个保证，即数据存储在w节点的地方。不能保证r节点的读取直到提示已经完成。

在所有常见的Dynamo实现中，松散法定人数是可选的。在Riak中，它们默认是启用的，而在Cassandra和Voldemort中它们默认是禁用的【46,49,50】。

运维多个数据中心

我们先前讨论了跨数据中心复制作为多主复制的用例（参阅“多主复制”）。无主复制还适用于多数据中心操作，因为它旨在容忍冲突的并发写入，网络中断和延迟尖峰。

Cassandra和Voldemort在正常的无主模型中实现了他们的多数据中心支持：副本的数量n包括所有数据中心的节点，在配置中，您可以指定每个数据中心中您想拥有的副本的数量。无论数据中心如何，每个来自客户端的写入都会发送到所有副本，但客户端通常只等待来自其本地数据中心内的法定节点的确认，从而不会受到跨数据中心链路延迟和中断的影响。对其他数据中心的高延迟写入通常被配置为异步发生，尽管配置有一定的灵活性【50,51】。

Riak将客户端和数据库节点之间的所有通信保持在一个数据中心本地，因此n描述了一个数据中心内的副本数量。数据库集群之间的跨数据中心复制在后台异步发生，其风格类似于多领导者复制【52】。

检测并发写入

Dynamo风格的数据库允许多个客户端同时写入相同的Key，这意味着即使使用严格的法定人数也会发生冲突。这种情况与多领导者复制相似（参阅“处理写入冲突”），但在Dynamo样式的数据库中，在读修复或带提示的接力期间也可能会产生冲突。

问题在于，由于可变的网络延迟和部分故障，事件可能在不同的节点以不同的顺序到达。例如，图5-12显示了两个客户机A和B同时写入三节点数据存储区中的键X：

- 节点 1 接收来自 A 的写入，但由于暂时中断，从不接收来自 B 的写入。
- 节点 2 首先接收来自 A 的写入，然后接收来自 B 的写入。
- 节点 3 首先接收来自 B 的写入，然后从 A 写入。

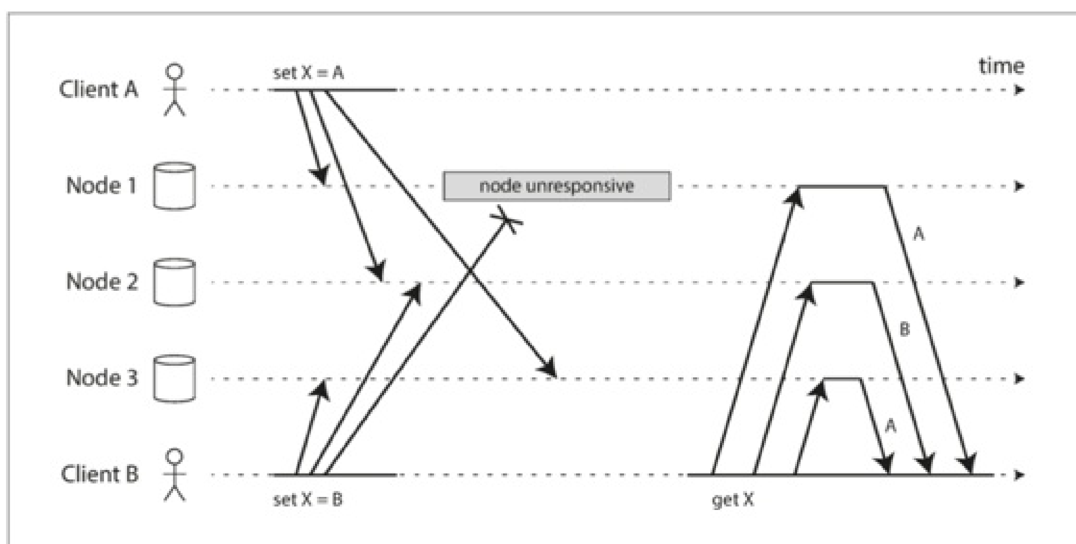


图**5-12** 并发写入**Dynamo**风格的数据存储：没有明确定义的顺序。

如果每个节点只要接收到来自客户端的写入请求就简单地覆盖了某个键的值，那么节点就会永久地不一致，如图5-12中的最终获取请求所示：节点2认为 X 的最终值是 B，而其他节点认为值是 A 。

为了最终达成一致，副本应该趋于相同的值。如何做到这一点？有人可能希望复制的数据库能够自动处理，但不幸的是，大多数的实现都很糟糕：如果你想避免丢失数据，你（应用程序开发人员）需要知道很多有关数据库冲突处理的内部信息。

在“处理写冲突”一节中已经简要介绍了一些解决冲突的技术。在总结本章之前，让我们来更详细地探讨这个问题。

最后写入为准（丢弃并发写入）

实现最终融合的一种方法是声明每个副本只需要存储最“最近”的值，并允许“更旧”的值被覆盖和抛弃。然后，只要我们有一种明确的方式来确定哪个写是“最近的”，并且每个写入最终都被复制到每个副本，那么复制最终会收敛到相同的值。

正如“最近”的引号所表明的，这个想法其实颇具误导性。在图5-12的例子中，当客户端向数据库节点发送写入请求时，客户端都不知道另一个客户端，因此不清楚哪一个先发生了。事实上，说“发生”是没有意义的：我们说写入是并发（**concurrent**）的，所以它们的顺序是不确定的。

即使写入没有自然的排序，我们也可以强制任意排序。例如，可以为每个写入附加一个时间戳，挑选最“最近”的最大时间戳，并丢弃具有较早时间戳的任何写入。这种冲突解决算法被称为最后写入为准（**LWW, last write wins**），是Cassandra 【53】唯一支持的冲突解决方法，也是Riak 【35】中的一个可选特征。

LWW实现了最终收敛的目标，但以持久性为代价：如果同一个Key有多个并发写入，即使它们都被报告为客户端成功（因为它们被写入 w 个副本），其中一个写道会生存下来，其他的将被无声丢弃。此外，LWW甚至可能会删除不是并发的写入，我们将在的“有序事件的时间

戳”中讨论。

有一些情况，如缓存，其中丢失的写入可能是可以接受的。如果丢失数据不可接受，LWW是解决冲突的一个很烂的选择。

与LWW一起使用数据库的唯一安全方法是确保一个键只写入一次，然后视为不可变，从而避免对同一个密钥进行并发更新。例如，推荐使用Cassandra的方法是使用UUID作为键，从而为每个写操作提供一个唯一的键【53】。

“此前发生”的关系和并发

我们如何判断两个操作是否是并发的？为了建立一个直觉，让我们看看一些例子：

- 在图5-9中，两个写入不是并发的：A的插入发生在B的增量之前，因为B递增的值是A插入的值。换句话说，B的操作建立在A的操作上，所以B的操作必须有后来发生。我们也可以说B是因果依赖（**causally dependent**）于A
- 另一方面，图5-12中的两个写入是并发的：当每个客户端启动操作时，它不知道另一个客户端也正在执行操作同样的Key。因此，操作之间不存在因果关系。

如果操作B了解操作A，或者依赖于A，或者以某种方式构建于操作A之上，则操作A在另一个操作B之前发生。在另一个操作之前是否发生一个操作是定义什么并发的关键。事实上，我们可以简单地说，如果两个操作都不在另一个之前发生，那么两个操作是并发的（即，两个操作都不知道另一个）【54】。

因此，只要有两个操作A和B，就有三种可能性：A在B之前发生，或者B在A之前发生，或者A和B并发。我们需要的是一个算法来告诉我们两个操作是否是并发的。如果一个操作发生在另一个操作之前，则后面的操作应该覆盖较早的操作，但是如果这些操作是并发的，则存在需要解决的冲突。

> 并发性，时间和相对性
>
> 如果两个操作“同时”发生，似乎应该称为并发——但事实上，它们在字面时间上重叠与否并不重要。由于分布式系统中的时钟问题，现实中是很难判断两个事件是否同时发生的，这个问题我们将在第8章中详细讨论。
>
> 为了定义并发性，确切的时间并不重要：如果两个操作都意识不到对方的存在，就称这两个操作并发，而不管它们发生的物理时间。人们有时把这个原理和狭义相对论的物理学联系起来【54】，它引入了信息不能比光速更快的思想。因此，如果事件之间的时间短于光通过它们之间的距离，那么发生一定距离的两个事件不可能相互影响。
>
> 在计算机系统中，即使光速原则上允许一个操作影响另一个操作，但两个操作也可能是并行的。例如，如果网络缓慢或中断，两个操作间可能会出现一段时间间隔，且仍然是并发的，因为网络问题阻止一个操作意识到另一个操作的存在。

捕获"此前发生"关系

来看一个算法，它确定两个操作是否为并发的，还是一个在另一个之前。为了简单起见，我们从一个只有一个副本的数据库开始。一旦我们已经制定了如何在单个副本上完成这项工作，我们可以将该方法概括为具有多个副本的无领导者数据库。

图5-13显示了两个客户端同时向同一购物车添加项目。（如果这样的例子让你觉得太麻烦了，那么可以想象，两个空中交通管制员同时把飞机添加到他们正在跟踪的区域）最初，购物车是空的。在它们之间，客户端向数据库发出五次写入：

1. 客户端 1 将牛奶加入购物车。这是该键的第一次写入，服务器成功存储了它并为其分配版本号1，最后将值与版本号一起回送给客户端。
2. 客户端 2 将鸡蛋加入购物车，不知道客户端 1 同时添加了牛奶（客户端 2 认为它的鸡蛋是购物车中的唯一物品）。服务器为此写入分配版本号 2，并将鸡蛋和牛奶存储为两个单独的值。然后它将这两个值都反回给客户端 2 ，并附上版本号 2 。
3. 客户端 1 不知道客户端 2 的写入，想要将面粉加入购物车，因此认为当前的购物车内容应该是 [牛奶，面粉]。它将此值与服务器先前向客户端 1 提供的版本号 1 一起发送到服务器。服务器可以从版本号中知道[牛奶，面粉]的写入取代了[牛奶]的先前值，但与[鸡蛋]的值是并发的。因此，服务器将版本 3 分配给[牛奶，面粉]，覆盖版本1值[牛奶]，但保留版本 2 的值[蛋]，并将所有的值返回给客户端 1 。
4. 同时，客户端 2 想要加入火腿，不知道客端户 1 刚刚加了面粉。客户端 2 在最后一个响应中从服务器收到了两个值[牛奶]和[蛋]，所以客户端 2 现在合并这些值，并添加火腿形成一个新的值，[鸡蛋，牛奶，火腿]。它将这个值发送到服务器，带着之前的版本号 2 。服务器检测到新值会覆盖版本 2 [eggs]，但新值也会与版本 3 [牛奶，面粉]并发，所以剩下的两个值是v3 [milk，flour]，和v4：[鸡蛋，牛奶，火腿]。
5. 最后，客户端 1 想要加培根。它以前在v3中从服务器接收[牛奶，面粉]和[鸡蛋]，所以它合并这些，添加培根，并将最终值[牛奶，面粉，鸡蛋，培根]连同版本号v3发往服务器。这会覆盖v3[牛奶，面粉]（请注意[鸡蛋]已经在最后一步被覆盖），但与v4[鸡蛋，牛奶，火腿]并发，所以服务器保留这两个并发值。

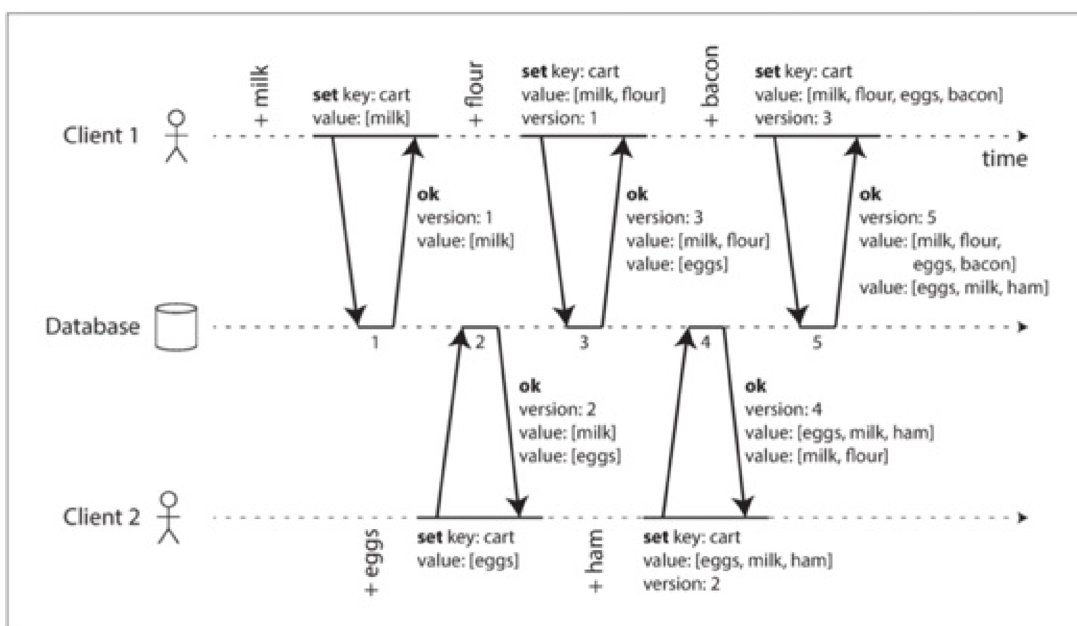


图**5-13** 捕获两个客户端之间的因果关系，同时编辑购物车。

图5-13中的操作之间的数据流如图5-14所示。 箭头表示哪个操作发生在其他操作之前，意味着后面的操作知道或依赖于较早的操作。 在这个例子中，客户端永远不会完全掌握服务器上的数据，因为总是有另一个操作同时进行。 但是，旧版本的值最终会被覆盖，并且不会丢失任何写入。

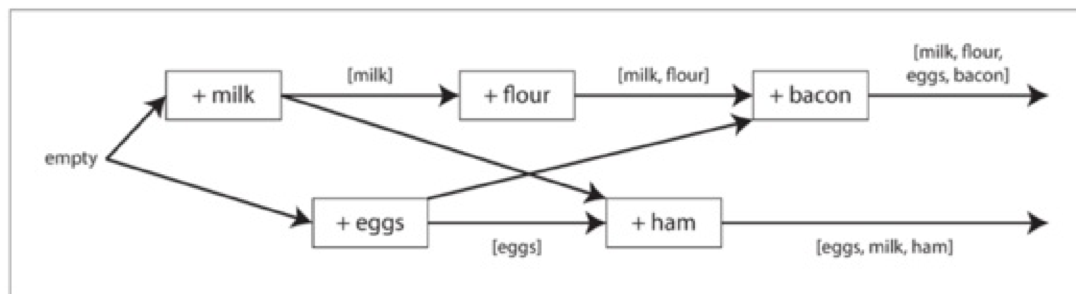


图**5-14** 图**5-13**中的因果依赖关系图。

请注意，服务器可以通过查看版本号来确定两个操作是否是并发的——它不需要解释该值本身（因此该值可以是任何数据结构）。该算法的工作原理如下：

- 服务器为每个键保留一个版本号，每次写入键时都增加版本号，并将新版本号与写入的值一起存储。
- 当客户端读取键时，服务器将返回所有未覆盖的值以及最新的版本号。客户端在写入前必须读取。
- 客户端写入键时，必须包含之前读取的版本号，并且必须将之前读取的所有值合并在一起。（来自写入请求的响应可以像读取一样，返回所有当前值，这使得我们可以像购物车示例那样连接多个写入。）
- 当服务器接收到具有特定版本号的写入时，它可以覆盖该版本号或更低版本的所有值（因为它知道它们已经被合并到新的值中），但是它必须保持所有值更高版本号（因为这些值与传入的写入同时发生）。

当一个写入包含前一次读取的版本号时，它会告诉我们写入的是哪一种状态。如果在不包含版本号的情况下进行写操作，则与所有其他写操作并发，因此它不会覆盖任何内容 —— 只会在随后的读取中作为其中一个值返回。

合并同时写入的值

这种算法可以确保没有数据被无声地丢弃，但不幸的是，客户端需要做一些额外的工作：如果多个操作并发发生，则客户端必须通过合并并发写入的值来擦屁股。 Riak称这些并发值兄弟（**siblings**）。

合并兄弟值，本质上是与多领导者复制中的冲突解决相同的问题，我们先前讨论过（参阅“处理写入冲突”）。一个简单的方法是根据版本号或时间戳（最后写入胜利）选择一个值，但这意味着丢失数据。所以，你可能需要在应用程序代码中做更聪明的事情。

以购物车为例，一种合理的合并兄弟方法就是集合求并。在图5-14中，最后的两个兄弟是[牛奶，面粉，鸡蛋，熏肉]和[鸡蛋，牛奶，火腿]。注意牛奶和鸡蛋出现在两个，即使他们每个只写一次。合并的价值可能是像[牛奶，面粉，鸡蛋，培根，火腿]，没有重复。

然而，如果你想让人们也可以从他们的手推车中删除东西，而不是仅仅添加东西，那么把兄弟求并可能不会产生正确的结果：如果你合并了两个兄弟手推车，并且只在其中一个兄弟值里删掉了它，那么被删除的项目会重新出现在兄弟的并集中【37】。为了防止这个问题，一个项目在删除时不能简单地从数据库中删除;相反，系统必须留下一个具有合适版本号的标记，以指示合并兄弟时该项目已被删除。这种删除标记被称为墓碑（**tombstone**）。（我们之前在“哈希索引”中的日志压缩的上下文中看到了墓碑。）

因为在应用程序代码中合并兄弟是复杂且容易出错的，所以有一些数据结构被设计出来用于自动执行这种合并，如“自动冲突解决”中讨论的。例如，Riak的数据类型支持使用称为CRDT的数据结构家族【38,39,55】可以以合理的方式自动合并兄弟，包括保留删除。

版本向量

图5-13中的示例只使用一个副本。如果有没有主库，有多个副本，算法如何改变？

图5-13使用单个版本号来捕获操作之间的依赖关系，但是当多个副本并发接受写入时，这是不够的。相反，除了对每个键使用版本号之外，还需要在每个副本中版本号。每个副本在处理写入时增加自己的版本号，并且跟踪从其他副本中看到的版本号。这个信息指出了要覆盖哪些值，以及保留哪些值作为兄弟。

所有副本的版本号集合称为版本向量（**version vector**）【56】。这个想法的一些变体正在使用，但最有趣的可能是在Riak 2.0 【58,59】中使用的分散版本矢量（**dotted version vector**）【57】。我们不会深入细节，但是它的工作方式与我们在购物车示例中看到的非常相似。

与图5-13中的版本号一样，当读取值时，版本向量会从数据库副本发送到客户端，并且随后写入值时需要将其发送回数据库。（Riak将版本向量编码为一个字符串，它称为因果上下文（**causal context**））。版本向量允许数据库区分覆盖写入和并发写入。

另外，就像在单个副本的例子中，应用程序可能需要合并兄弟。版本向量结构确保从一个副本读取并随后写回到另一个副本是安全的。这样做可能会创建兄弟，但只要兄弟姐妹合并正确，就不会丢失数据。

> 版本向量和向量时钟
>
> 版本向量有时也被称为矢量时钟，即使它们不完全相同。 差别很微妙——请参阅参考资料的细节【57,60,61】。 简而言之，在比较副本的状态时，版本向量是正确的数据结构。

本章小结

在本章中，我们考察了复制的问题。复制可以用于几个目的：

高可用性

即使在一台机器（或多台机器，或整个数据中心）停机的情况下也能保持系统正常运行

断开连接的操作

允许应用程序在网络中断时继续工作

延迟

将数据放置在距离用户较近的地方，以便用户能够更快地与其交互

可扩展性

能够处理比单个机器更高的读取量可以通过对副本进行读取来处理

尽管是一个简单的目标 - 在几台机器上保留相同数据的副本，但复制却是一个非常棘手的问题。它需要仔细考虑并发和所有可能出错的事情，并处理这些故障的后果。至少，我们需要处理不可用的节点和网络中断（甚至不考虑更隐蔽的故障，例如由于软件错误导致的无提示数据损坏）。

我们讨论了复制的三种主要方法：

单主复制

客户端将所有写入操作发送到单个节点（领导者），该节点将数据更改事件流发送到其他副本（追随者）。读取可以在任何副本上执行，但从追随者读取可能是陈旧的。

多主复制

客户端发送每个写入到几个领导节点之一，其中任何一个都可以接受写入。领导者将数据更改事件流发送给彼此以及任何跟随者节点。

无主复制

客户端发送每个写入到几个节点，并从多个节点并行读取，以检测和纠正具有陈旧数据的节点。 每种方法都有优点和缺点。单主复制是非常流行的，因为它很容易理解，不需要担心冲突解决。在出现故障节点，网络中断和延迟峰值的情况下，多领导者和无领导者复制可以更加稳健，但代价很难推理，只能提供非常弱的一致性保证。

复制可以是同步的，也可以是异步的，在发生故障时对系统行为有深远的影响。尽管在系统运行平稳时异步复制速度很快，但是在复制滞后增加和服务器故障时要弄清楚会发生什么，这一点很重要。如果一个领导者失败了，并且你推动一个异步更新的追随者成为新的领导者，那么最近承诺的数据可能会丢失。

我们研究了一些可能由复制滞后引起的奇怪效应，我们讨论了一些有助于决定应用程序在复制滞后时的行为的一致性模型：

写后读

用户应该总是看到自己提交的数据。

单调读

当用户在某个时间点看到数据后，他们不应该在较早的时间点看到数据。

一致前缀读

用户应该将数据视为具有因果意义的状态：例如，按照正确的顺序查看问题及其答复。

最后，我们讨论了多领导者和无领导者复制方法所固有的并发问题：因为他们允许多个写入并发发生冲突。我们研究了一个数据库可能使用的算法来确定一个操作是否发生在另一个操作之前，或者它们是否同时发生。我们还谈到了通过合并并发更新来解决冲突的方法。

在下一章中，我们将继续研究分布在多个机器上的数据，通过复制的对应方式：将大数据集分割成分区。

参考文献

1. Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: “Notes on Distributed Databases,” IBM Research, Research Report RJ2571(33471), July 1979.
2. “Oracle Active Data Guard Real-Time Data Protection and Availability,” Oracle White Paper, June 2013.
3. “AlwaysOn Availability Groups,” in *SQL Server Books Online*, Microsoft, 2012.
4. Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: “On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform,” at *ACM International Conference on Management of Data* (SIGMOD), June 2013.
5. Jun Rao: “Intra-Cluster Replication for Apache Kafka,” at *ApacheCon North America*, February 2013.
6. “Highly Available Queues,” in *RabbitMQ Server Documentation*, Pivotal Software, Inc., 2014.
7. Yoshinori Matsunobu: “Semi-Synchronous Replication at Facebook,” *yoshinorimatsunobu.blogspot.co.uk*, April 1, 2014.
8. Robbert van Renesse and Fred B. Schneider: “Chain Replication for Supporting High Throughput and Availability,” at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

9. Jeff Terrace and Michael J. Freedman: "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads," at *USENIX Annual Technical Conference* (ATC), June 2009.
10. Brad Calder, Ju Wang, Aaron Ogus, et al.: "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," at *23rd ACM Symposium on Operating Systems Principles* (SOSP), October 2011.
11. Andrew Wang: "Windows Azure Storage," *umbrant.com*, February 4, 2016.
12. "Percona Xtrabackup - Documentation," Percona LLC, 2014.
13. Jesse Newland: "GitHub Availability This Week," *github.com*, September 14, 2012.
14. Mark Imbriaco: "Downtime Last Saturday," *github.com*, December 26, 2012.
15. John Hugg: "'All in' with Determinism for Performance and Testing in Distributed Systems," at *Strange Loop*, September 2015. Amit Kapila: "WAL Internals of PostgreSQL," at *PostgreSQL Conference* (PGCon), May 2012.
16. *MySQL Internals Manual*. Oracle, 2014.
17. Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: "Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.
18. "Oracle GoldenGate 12c: Real-Time Access to Real-Time Information," Oracle White Paper, October 2013.
19. Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *ACM Symposium on Cloud Computing* (SoCC), October 2012.
20. Greg Sabino Mullane: "Version 5 of Bucardo Database Replication System," *blog.endpoint.com*, June 23, 2014.
21. Werner Vogels: "Eventually Consistent," *ACM Queue*, volume 6, number 6, pages 14–19, October 2008. doi:10.1145/1466443.1466448
22. Douglas B. Terry: "Replicated Data Consistency Explained Through Baseball," Microsoft Research, Technical Report MSR-TR-2011-137, October 2011.
23. Douglas B. Terry, Alan J. Demers, Karin Petersen, et al.: "Session Guarantees for Weakly Consistent Replicated Data," at *3rd International Conference on Parallel and Distributed Information Systems* (PDIS), September 1994. doi:10.1109/PDIS.1994.331722

24. Terry Pratchett: *Reaper Man: A Discworld Novel*. Victor Gollancz, 1991. ISBN: 978-0-575-04979-6

25. “Tungsten Replicator,” Continuent, Inc., 2014.

26. “BDR 0.10.0 Documentation,” The PostgreSQL Global Development Group, *bdr-project.org*, 2015.

27. Robert Hodges: “If You *Must* Deploy Multi-Master Replication, Read This First,” *scale-out-blog.blogspot.co.uk*, March 30, 2012.

28. J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6

29. AppJet, Inc.: “Etherpad and EasySync Technical Manual,” *github.com*, March 26, 2011.

30. John Day-Richter: “What’s Different About the New Google Docs: Making Collaboration Fast,” *googledrive.blogspot.com*, 23 September 2010.

31. Martin Kleppmann and Alastair R. Beresford: “A Conflict-Free Replicated JSON Datatype,” arXiv:1608.03960, August 13, 2016.

32. Frazer Clement: “Eventual Consistency – Detecting Conflicts,” *messagepassing.blogspot.co.uk*, October 20, 2011.

33. Robert Hodges: “State of the Art for MySQL Multi-Master Replication,” at *Percona Live: MySQL Conference & Expo*, April 2013.

34. John Daily: “Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems,” *basho.com*, November 12, 2013.

35. Riley Berton: “Is Bi-Directional Replication (BDR) in Postgres Transactional?,” *sdf.org*, January 4, 2016.

36. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: “Dynamo: Amazon's Highly Available Key-Value Store,” at *21st ACM Symposium on Operating Systems Principles* (SOSP), October 2007.

37. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski: “A Comprehensive Study of Convergent and Commutative Replicated Data Types,” INRIA Research Report no. 7506, January 2011.

38. Sam Elliott: “CRDTs: An UPDATE (or Maybe Just a PUT),” at *RICON West*, October 2013.

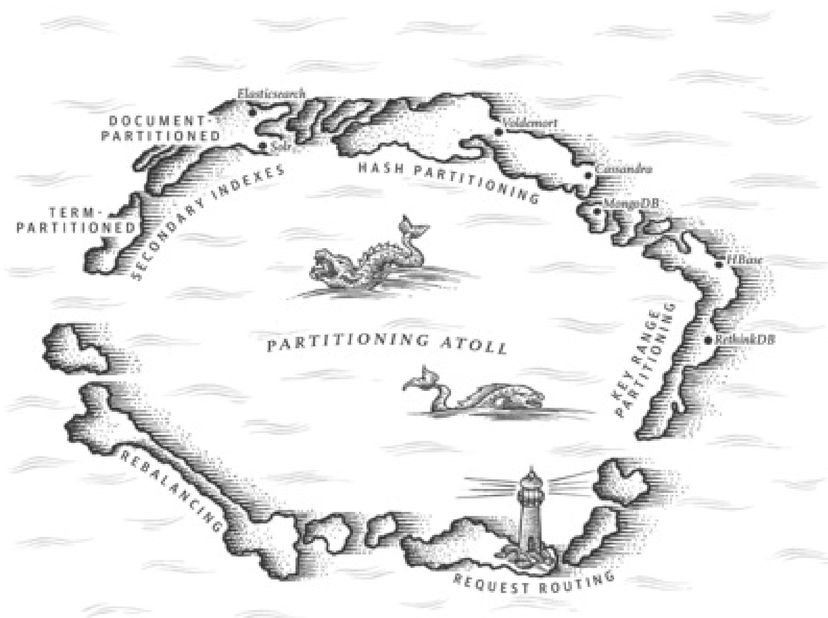
39. Russell Brown: “A Bluffers Guide to CRDTs in Riak,” *gist.github.com*, October 28, 2013.

40. Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy: “Mergeable Persistent Data Structures,” at *26es Journées Francophones des Langages Applicatifs* (JFLA), January 2015.
41. Chengzheng Sun and Clarence Ellis: “Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements,” at *ACM Conference on Computer Supported Cooperative Work* (CSCW), November 1998.
42. Lars Hofhansl: “HBASE-7709: Infinite Loop Possible in Master/Master Replication,” *issues.apache.org*, January 29, 2013.
43. David K. Gifford: “Weighted Voting for Replicated Data,” at *7th ACM Symposium on Operating Systems Principles* (SOSP), December 1979. doi:10.1145/800215.806583
44. Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: “Flexible Paxos: Quorum Intersection Revisited,” *arXiv:1608.06696*, August 24, 2016.
45. Joseph Blomstedt: “Re: Absolute Consistency,” email to *riak-users* mailing list, *lists.basho.com*, January 11, 2012.
46. Joseph Blomstedt: “Bringing Consistency to Riak,” at *RICON West*, October 2012.
47. Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, et al.: “Quantifying Eventual Consistency with PBS,” *Communications of the ACM*, volume 57, number 8, pages 93–102, August 2014. doi:10.1145/2632792
48. Jonathan Ellis: “Modern Hinted Handoff,” *datastax.com*, December 11, 2012.
49. “Project Voldemort Wiki,” *github.com*, 2013.
50. “Apache Cassandra 2.0 Documentation,” DataStax, Inc., 2014.
51. “Riak Enterprise: Multi-Datacenter Replication.” Technical whitepaper, Basho Technologies, Inc., September 2014.
52. Jonathan Ellis: “Why Cassandra Doesn't Need Vector Clocks,” *datastax.com*, September 2, 2013.
53. Leslie Lamport: “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563
54. Joel Jacobson: “Riak 2.0: Data Types,” *blog.joeljacobson.com*, March 23, 2014.
55. D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, et al.: “Detection of Mutual Inconsistency in Distributed Systems,” *IEEE Transactions on Software Engineering*, volume 9, number 3, pages 240–247, May 1983. doi:10.1109/TSE.1983.236733

56. Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, et al.: “Dotted Version Vectors: Logical Clocks for Optimistic Replication,” arXiv:1011.5808, November 26, 2010.
57. Sean Cribbs: “A Brief History of Time in Riak,” at *RICON*, October 2014.
58. Russell Brown: “Vector Clocks Revisited Part 2: Dotted Version Vectors,” *basho.com*, November 10, 2015.
59. Carlos Baquero: “Version Vectors Are Not Vector Clocks,” *haslab.wordpress.com*, July 8, 2011.
60. Reinhard Schwarz and Friedemann Mattern: “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail,” *Distributed Computing*, volume 7, number 3, pages 149–174, March 1994. doi:10.1007/BF02277859

上一章	目录	下一章
第二部分：分布式数据	设计数据密集型应用	第六章：分区

6. 分区



> 我们必须跳出电脑指令序列的窠臼。 叙述定义、描述元数据、梳理关系，而不是编写过程。
>
> —— Grace Murray Hopper，未来的计算机及其管理（1962）

[TOC]

在第5章中，我们讨论了复制——即数据在不同节点上的副本,对于非常大的数据集,或非常高的吞吐量，仅仅进行复制是不够的：我们需要将数据进行分区（**partitions**），也称为分片（**sharding**）[i]

> i. 正如本章所讨论的，分区是一种有意将大型数据库分解成小型数据库的方式。它与网络分区（**net splits**）无关，这是节点之间网络中的一种故障类型。我们将在第8章讨论这些错误。 ↩

> 术语澄清
>
> 上文中的分区(**partition**),在MongoDB,Elasticsearch和Solr Cloud中被称为分片(**shard**),在HBase中称之为区域(**Region**)，Bigtable中则是 表块（**tablet**），Cassandra和Riak中是虚节点（**vnode**), Couchbase中叫做虚桶(**vBucket**).但是分区(**partition**) 是约定俗成的叫法。

通常情况下，每条数据（每条记录，每行或每个文档）属于且仅属于一个分区。有很多方法可以实现这一点，本章将进行深入讨论。实际上，每个分区都是自己的小型数据库，尽管数据库可能支持同时进行多个分区的操作。

分区主要是为了可扩展性。不同的分区可以放在不共享集群中的不同节点上（参阅第二部分关于无共享架构的定义）。因此，大数据集可以分布在多个磁盘上，并且查询负载可以分布在多个处理器上。

对于在单个分区上运行的查询，每个节点可以独立执行对自己的查询，因此可以通过添加更多的节点来扩大查询吞吐量。大型，复杂的查询可能会跨越多个节点并行处理，尽管这也带来了新的困难。

分区数据库在20世纪80年代由Teradata和NonStop SQL【1】等产品率先推出，最近因为NoSQL数据库和基于Hadoop的数据仓库重新被关注。有些系统是为事务性工作设计的，有些系统则用于分析（参阅“[事务处理或分析]”）：这种差异会影响系统的运作方式，但是分区的基本原理均适用于这两种工作方式。

在本章中，我们将首先介绍分割大型数据集的不同方法，并观察索引如何与分区配合。然后我们将讨论重新平衡分区，如果想要添加或删除群集中的节点，则必须进行再平衡。最后，我们将概述数据库如何将请求路由到正确的分区并执行查询。

分区与复制

分区通常与复制结合使用，使得每个分区的副本存储在多个节点上。 这意味着，即使每条记录属于一个分区，它仍然可以存储在多个不同的节点上以获得容错能力。

一个节点可能存储多个分区。 如果使用主从复制模型，则分区和复制的组合如图6-1所示。 每个分区领导者(主)被分配给一个节点，追随者(从)被分配给其他节点。 每个节点可能是某些分区的领导者，同时是其他分区的追随者。 我们在第5章讨论的关于数据库复制的所有内容同样适用于分区的复制。 大多数情况下，分区方案的选择与复制方案的选择是独立的，为简单起见，本章中将忽略复制。

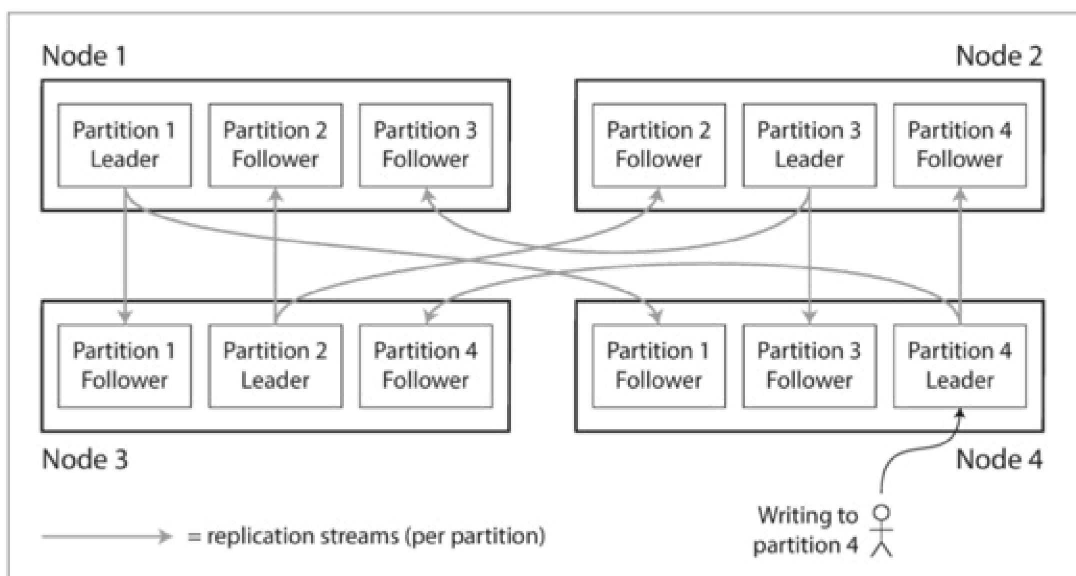


图**6-1** 组合使用复制和分区：每个节点充当某些分区的领导者，其他分区充当追随者。

键值数据的分区

假设你有大量数据并且想要分区,如何决定在哪些节点上存储哪些记录呢？

分区目标是将数据和查询负载均匀分布在各个节点上。如果每个节点公平分享数据和负载，那么理论上10个节点应该能够处理10倍的数据量和10倍的单个节点的读写吞吐量（暂时忽略复制）。

如果分区是不公平的，一些分区比其他分区有更多的数据或查询，我们称之为偏斜（**skew**）。数据偏斜的存在使分区效率下降很多。在极端的情况下，所有的负载可能压在一个分区上，其余9个节点空闲的，瓶颈落在这一个繁忙的节点上。不均衡导致的高负载的分区被称为热点（**hot spot**）。

避免热点最简单的方法是将记录随机分配给节点。这将在所有节点上平均分配数据，但是它有一个很大的缺点：当你试图读取一个特定的值时，你无法知道它在哪个节点上，所以你必须并行地查询所有的节点。

我们可以做得更好。现在假设您有一个简单的键值数据模型，其中您总是通过其主键访问记录。例如，在一本老式的纸质百科全书中，你可以通过标题来查找一个条目;由于所有条目按字母顺序排序，因此您可以快速找到您要查找的条目。

根据键的范围分区

一种分区的方法是为每个分区指定一块连续的键范围（从最小值到最大值），如纸百科全书的卷（图6-2）。如果知道范围之间的边界，则可以轻松确定哪个分区包含某个值。如果您还知道分区所在的节点，那么可以直接向相应的节点发出请求（对于百科全书而言，就像从书架上选取正确的书籍）。

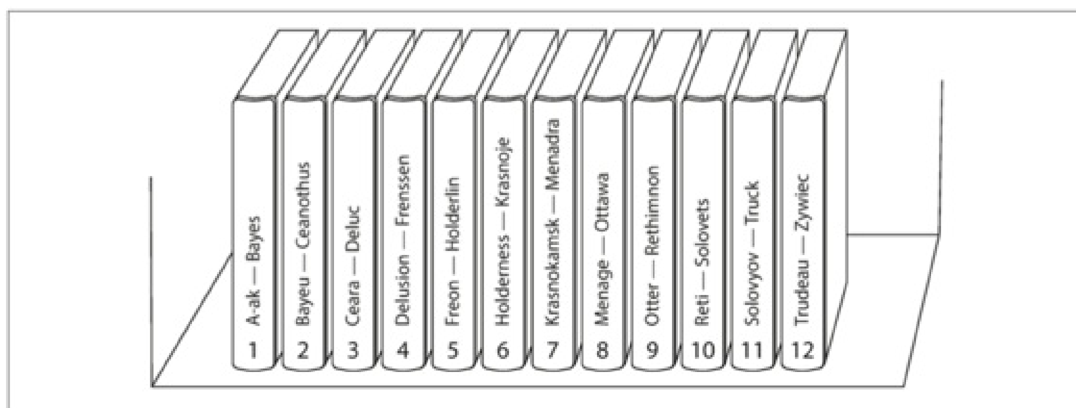


图**6-2** 印刷版百科全书按照关键字范围进行分区

键的范围不一定均匀分布，因为数据也很可能不均匀分布。例如在图6-2中，第1卷包含以A和B开头的单词，但第12卷则包含以T，U，V，X，Y和Z开头的单词。只是简单的规定每个卷包含两个字母会导致一些卷比其他卷大。为了均匀分配数据，分区边界需要依据数据调整。

分区边界可以由管理员手动选择，也可以由数据库自动选择（我们会在“重新平衡分区”中更详细地讨论分区边界的选择）。Bigtable使用了这种分区策略，以及其开源等价物HBase【2, 3】，RethinkDB和2.4版本之前的MongoDB【4】。

在每个分区中，我们可以按照一定的顺序保存键（参见“SSTables和LSM-树”）。好处是进行范围扫描非常简单，您可以将键作为联合索引来处理，以便在一次查询中获取多个相关记录（参阅“多列索引”）。例如，假设我们有一个程序来存储传感器网络的数据，其中主键是测量的时间戳（年月日时分秒）。范围扫描在这种情况下非常有用，因为我们可以轻松获取某个月份的所有数据。

然而，Key Range分区的缺点是某些特定的访问模式会导致热点。 如果主键是时间戳，则分区对应于时间范围，例如，给每天分配一个分区。 不幸的是，由于我们在测量发生时将数据从传感器写入数据库，因此所有写入操作都会转到同一个分区（即今天的分区），这样分区可能会因写入而过载，而其他分区则处于空闲状态【5】。

为了避免传感器数据库中的这个问题，需要使用除了时间戳以外的其他东西作为主键的第一个部分。 例如，可以在每个时间戳前添加传感器名称，这样会首先按传感器名称，然后按时间进行分区。 假设有多个传感器同时运行，写入负载将最终均匀分布在不同分区上。 现在，当想要在一个时间范围内获取多个传感器的值时，您需要为每个传感器名称执行一个单独的范围查询。

根据键的散列分区

由于偏斜和热点的风险，许多分布式数据存储使用散列函数来确定给定键的分区。

一个好的散列函数可以将将偏斜的数据均匀分布。假设你有一个32位散列函数,无论何时给定一个新的字符串输入，它将返回一个0到2^{32} -1之间的"随机"数。即使输入的字符串非常相似，它们的散列也会均匀分布在这个数字范围内。

出于分区的目的，散列函数不需要多么强壮的加密算法：例如，Cassandra和MongoDB使用MD5，Voldemort使用Fowler-Noll-Vo函数。许多编程语言都有内置的简单哈希函数（它们用于哈希表），但是它们可能不适合分区：例如，在Java的 `Object.hashCode()` 和Ruby的 `Object#hash` ，同一个键可能在不同的进程中有不同的哈希值【6】。

一旦你有一个合适的键散列函数，你可以为每个分区分配一个散列范围（而不是键的范围），每个通过哈希散列落在分区范围内的键将被存储在该分区中。如图6-3所示。

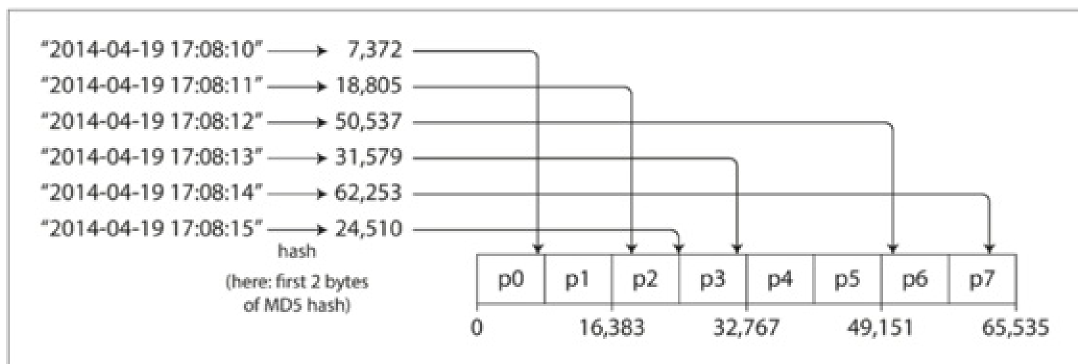


图**6-3** 按哈希键分区

这种技术擅长在分区之间分配键。分区边界可以是均匀间隔的，也可以是伪随机选择的（在这种情况下，该技术有时也被称为一致性哈希（**consistent hashing**））。

> 一致性哈希
>
> 一致性哈希由Karger等人定义。【7】 用于跨互联网级别的缓存系统，例如CDN中，是一种能均匀分配负载的方法。它使用随机选择的分区边界（**partition boundaries**）来避免中央控制或分布式一致性的需要。 请注意，这里的一致性与复制一致性（请参阅第5章）或ACID一致性（参阅第7章）无关，而是描述了重新平衡的特定方法。
>
> 正如我们将在“重新平衡分区”中所看到的，这种特殊的方法对于数据库实际上并不是很好，所以在实际中很少使用（某些数据库的文档仍然指的是一致性哈希，但是它 往往是不准确的）。 因为有可能产生混淆，所以最好避免使用一致性哈希这个术语，而只是把它称为散列分区（**hash partitioning**）。

不幸的是，通过使用Key散列进行分区，我们失去了键范围分区的一个很好的属性：高效执行范围查询的能力。曾经相邻的密钥现在分散在所有分区中，所以它们之间的顺序就丢失了。在MongoDB中，如果您使用了基于散列的分区模式，则任何范围查询都必须发送到所有分区【4】。Riak 【9】，Couchbase 【10】或Voldemort不支持主键上的范围查询。

Cassandra采取了折衷的策略【11, 12, 13】。 Cassandra中的表可以使用由多个列组成的复合主键来声明。键中只有第一列会作为散列的依据，而其他列则被用作Casssandra的SSTables中排序数据的连接索引。尽管查询无法在复合主键的第一列中按范围扫表，但如果第一列已经指定了固定值，则可以对该键的其他列执行有效的范围扫描。

组合索引方法为一对多关系提供了一个优雅的数据模型。例如，在社交媒体网站上，一个用户可能会发布很多更新。如果更新的主键被选择为 `(user_id, update_timestamp)` ，那么您可以有效地检索特定用户在某个时间间隔内按时间戳排序的所有更新。不同的用户可以存储在不同的分区上，对于每个用户，更新按时间戳顺序存储在单个分区上。

负载倾斜与消除热点

如前所述，哈希分区可以帮助减少热点。但是，它不能完全避免它们：在极端情况下，所有的读写操作都是针对同一个键的，所有的请求都会被路由到同一个分区。

这种场景也许并不常见，但并非闻所未闻：例如，在社交媒体网站上，一个拥有数百万追随者的名人用户在做某事时可能会引发一场风暴【14】。这个事件可能导致大量写入同一个键（键可能是名人的用户ID，或者人们正在评论的动作的ID）。哈希策略不起作用，因为两个相同ID的哈希值仍然是相同的。

如今，大多数数据系统无法自动补偿这种高度偏斜的负载，因此应用程序有责任减少偏斜。例如，如果一个主键被认为是非常火爆的，一个简单的方法是在主键的开始或结尾添加一个随机数。只要一个两位数的十进制随机数就可以将主键分散为100钟不同的主键,从而存储在不同的分区中。

然而，将主键进行分割之后，任何读取都必须要做额外的工作，因为他们必须从所有100个主键分布中读取数据并将其合并。此技术还需要额外的记录：只需要对少量热点附加随机数;对于写入吞吐量低的绝大多数主键来是不必要的开销。因此，您还需要一些方法来跟踪哪些键需要被分割。

也许在将来，数据系统将能够自动检测和补偿偏斜的工作负载；但现在，您需要自己来权衡。

分片与次级索引

到目前为止，我们讨论的分区方案依赖于键值数据模型。如果只通过主键访问记录，我们可以从该键确定分区，并使用它来将读写请求路由到负责该键的分区。

如果涉及次级索引，情况会变得更加复杂（参考“其他索引结构”）。辅助索引通常并不能唯一地标识记录，而是一种搜索记录中出现特定值的方式：查找用户123的所有操作，查找包含词语 `hogwash` 的所有文章，查找所有颜色为红色的车辆等等。

次级索引是关系型数据库的基础，并且在文档数据库中也很普遍。许多键值存储（如HBase和Volde-mort）为了减少实现的复杂度而放弃了次级索引，但是一些（如Riak）已经开始添加它们，因为它们对于数据模型实在是太有用了。并且次级索引也是Solr和Elasticsearch等搜索服务器的基石。

次级索引的问题是它们不能整齐地映射到分区。有两种用二级索引对数据库进行分区的方法：基于文档的分区（**document-based**）和基于关键词（**term-based**）的分区。

按文档的二级索引

假设你正在经营一个销售二手车的网站（如图6-4所示）。 每个列表都有一个唯一的ID——称之为文档ID——并且用文档ID对数据库进行分区（例如，分区0中的ID 0到499，分区1中的ID 500到999等）。

你想让用户搜索汽车，允许他们通过颜色和厂商过滤，所以需要一个在颜色和厂商上的次级索引（文档数据库中这些是字段（**field**），关系数据库中这些是列（**column**））。 如果您声明了索引，则数据库可以自动执行索引[ii]。例如，无论何时将红色汽车添加到数据库，数据库分区都会自动将其添加到索引条目 `color：red` 的文档ID列表中。

> [ii]. 如果数据库仅支持键值模型，则你可能会尝试在应用程序代码中创建从值到文档ID的映射来实现辅助索引。 如果沿着这条路线走下去，请万分小心，确保您的索引与底层数据保持一致。 竞争条件和间歇性写入失败（其中一些更改已保存，但其他更改未保存）很容易导致数据不同步 - 参见“多对象事务的需求”。 ↩

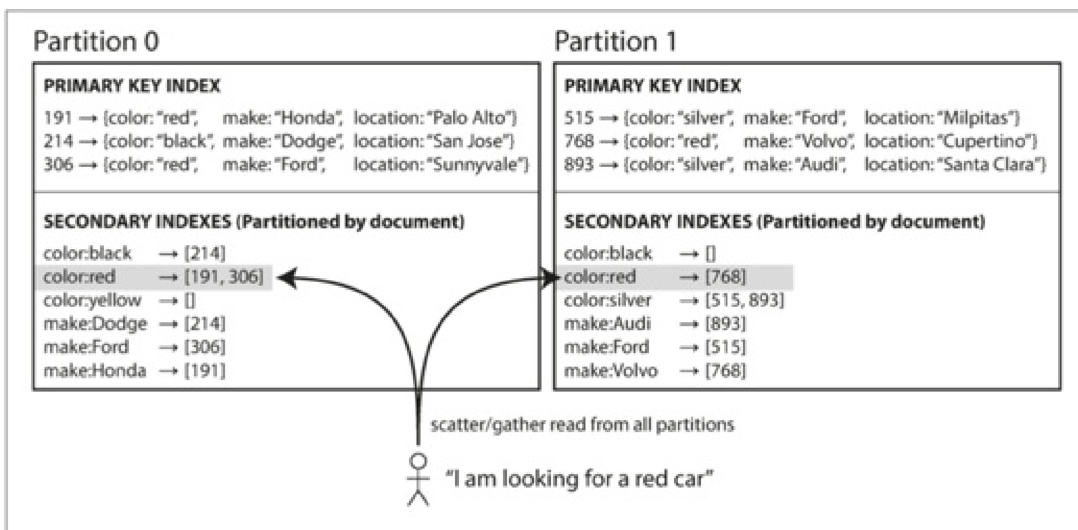


图**6-4** 按文档分区二级索引

在这种索引方法中，每个分区是完全独立的：每个分区维护自己的二级索引，仅覆盖该分区中的文档。它不关心存储在其他分区的数据。无论何时您需要写入数据库（添加，删除或更新文档），只需处理包含您正在编写的文档ID的分区即可。出于这个原因，文档分区索引也被称为本地索引（**local index**）（而不是将在下一节中描述的全局索引（**global index**））。

但是，从文档分区索引中读取需要注意：除非您对文档ID做了特别的处理，否则没有理由将所有具有特定颜色或特定品牌的汽车放在同一个分区中。在图6-4中，红色汽车出现在分区0和分区1中。因此，如果要搜索红色汽车，则需要将查询发送到所有分区，并合并所有返回的结果。

这种查询分区数据库的方法有时被称为分散/聚集（**scatter/gather**），并且可能会使二级索引上的读取查询相当昂贵。即使并行查询分区，分散/聚集也容易导致尾部延迟放大（参阅“实践中的百分位点”）。然而，它被广泛使用：MonDBDB，Riak 【15】，Cassandra 【16】，Elasticsearch 【17】，SolrCloud 【18】和VoltDB 【19】都使用文档分区二级索引。大多数数据库供应商建议您构建一个能从单个分区提供二级索引查询的分区方案，但这并不总是可行，尤其是当在单个查询中使用多个二级索引时（例如同时需要按颜色和制造商查询）。

根据关键词(Term)的二级索引

我们可以构建一个覆盖所有分区数据的全局索引，而不是给每个分区创建自己的次级索引（本地索引）。但是，我们不能只把这个索引存储在一个节点上，因为它可能会成为瓶颈，违背了分区的目的。全局索引也必须进行分区，但可以采用与主键不同的分区方式。

图6-5阐述了这可能是什么样子：来自所有分区的红色汽车在红色索引中，并且索引是分区的，首字母从 `a` 到 `r` 的颜色在分区0中， `s` 到 `z` 的在分区1。汽车制造商的索引也与之类似（分区边界在 `f` 和 `h` 之间）。

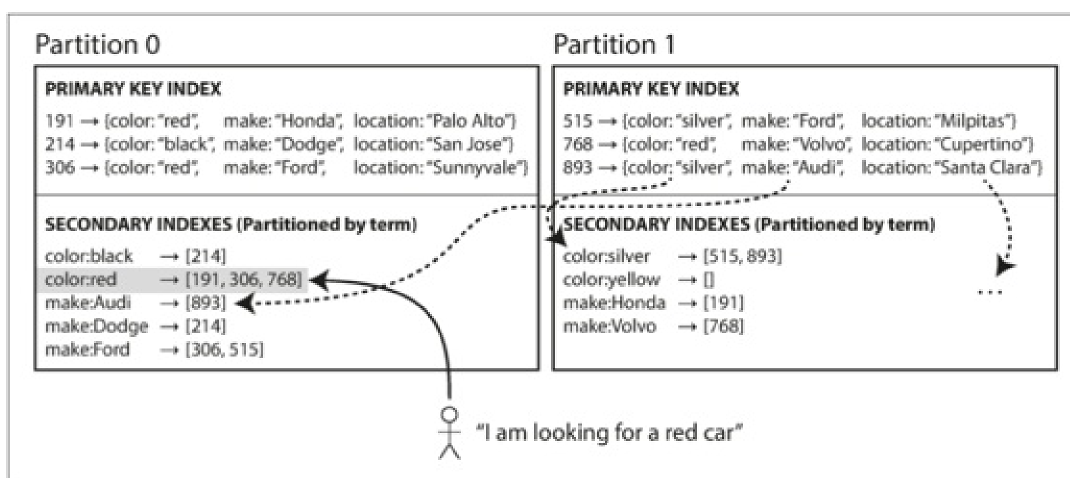


图**6-5** 按关键词对二级索引进行分区

我们将这种索引称为关键词分区（**term-partitioned**），因为我们寻找的关键词决定了索引的分区方式。例如，一个关键词可能是： `颜色：红色` 。关键词**(Term)** 来源于来自全文搜索索引（一种特殊的次级索引），指文档中出现的所有单词。

和之前一样，我们可以通过关键词本身或者它的散列进行索引分区。根据它本身分区对于范围扫描非常有用（例如对于数字,像汽车的报价），而对关键词的哈希分区提供了负载均衡的能力。

关键词分区的全局索引优于文档分区索引的地方点是它可以使读取更有效率：不需要分散/收集所有分区，客户端只需要向包含关键词的分区发出请求。全局索引的缺点在于写入速度较慢且较为复杂，因为写入单个文档现在可能会影响索引的多个分区（文档中的每个关键词可能位于不同的分区或者不同的节点上） 。

理想情况下，索引总是最新的，写入数据库的每个文档都会立即反映在索引中。但是，在关键词分区索引中，这需要跨分区的分布式事务，并不是所有数据库都支持（请参阅第7章和第9章）。

在实践中，对全局二级索引的更新通常是异步的（也就是说，如果在写入之后不久读取索引，刚才所做的更改可能尚未反映在索引中）。例如，Amazon DynamoDB声称在正常情况下，其全局次级索引会在不到一秒的时间内更新，但在基础架构出现故障的情况下可能会有延迟【20】。

全局关键词分区索引的其他用途包括Riak的搜索功能【21】和Oracle数据仓库，它允许您在本地和全局索引之间进行选择【22】。我们将在第12章中涉及实现关键字二级索引的话题。

分区再平衡

随着时间的推移，数据库会有各种变化。

- 查询吞吐量增加，所以您想要添加更多的CPU来处理负载。
- 数据集大小增加，所以您想添加更多的磁盘和RAM来存储它。
- 机器出现故障，其他机器需要接管故障机器的责任。

所有这些更改都需要数据和请求从一个节点移动到另一个节点。 将负载从集群中的一个节点向另一个节点移动的过程称为再平衡（**reblancing**）。

无论使用哪种分区方案，再平衡通常都要满足一些最低要求：

- 再平衡之后，负载（数据存储，读取和写入请求）应该在集群中的节点之间公平地共享。
- 再平衡发生时，数据库应该继续接受读取和写入。
- 节点之间只移动必须的数据，以便快速再平衡，并减少网络和磁盘I/O负载。

平衡策略

有几种不同的分区分配方法【23】,让我们依次简要讨论一下。

反面教材：hash mod N

我们在前面说过（图6-3），最好将可能的散列分成不同的范围，并将每个范围分配给一个分区（例如，如果$0≤hash(key)<b_0$，则将键分配给分区0，如果$b_0 ≤ hash(key) <b_1$，则分配给分区1）

也许你想知道为什么我们不使用***mod***（许多编程语言中的%运算符）。例如， `hash(key) mod 10` 会返回一个介于0和9之间的数字（如果我们将散列写为十进制数，散列模10将是最后一个数字）。如果我们有10个节点，编号为0到9，这似乎是将每个键分配给一个节点的简单方法。

模N方法的问题是，如果节点数量N发生变化，大多数密钥将需要从一个节点移动到另一个节点。例如，假设$hash(key)=123456$。如果最初有10个节点，那么这个键一开始放在节点6上（因为$123456\ mod\ 10 = 6$）。当您增长到11个节点时，密钥需要移动到节点3（$123456\ mod\ 11 = 3$），当您增长到12个节点时，需要移动到节点0（$123456\ mod\ 12 = 0$）。这种频繁的举动使得重新平衡过于昂贵。

我们需要一种只移动必需数据的方法。

固定数量的分区

幸运的是，有一个相当简单的解决方案：创建比节点更多的分区，并为每个节点分配多个分区。例如，运行在10个节点的集群上的数据库可能会从一开始就被拆分为1,000个分区，因此大约有100个分区被分配给每个节点。

现在，如果一个节点被添加到集群中，新节点可以从当前每个节点中窃取一些分区，直到分区再次公平分配。这个过程如图6-6所示。如果从集群中删除一个节点，则会发生相反的情况。

只有分区在节点之间的移动。分区的数量不会改变，键所指定的分区也不会改变。唯一改变的是分区所在的节点。这种变更并不是即时的 — 在网络上传输大量的数据需要一些时间 — 所以在传输过程中，原有分区仍然会接受读写操作。

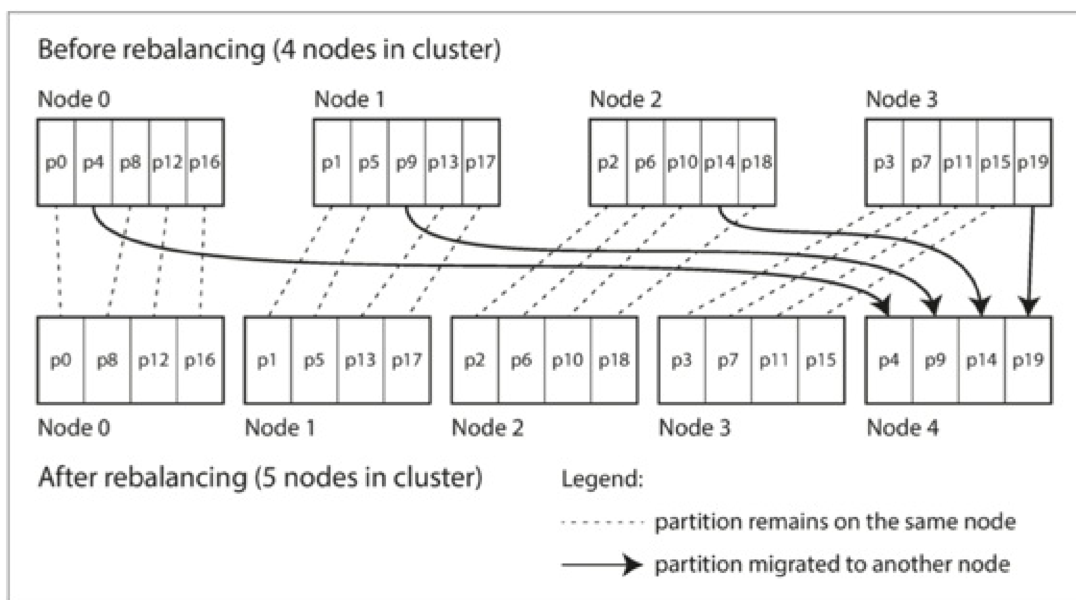


图**6-6** 将新节点添加到每个节点具有多个分区的数据库群集。

原则上，您甚至可以解决集群中的硬件不匹配问题：通过为更强大的节点分配更多的分区，可以强制这些节点承载更多的负载。在Riak 【15】，Elasticsearch 【24】，Couchbase 【10】和Voldemort 【25】中使用了这种再平衡的方法。

在这种配置中，分区的数量通常在数据库第一次建立时确定，之后不会改变。虽然原则上可以分割和合并分区（请参阅下一节），但固定数量的分区在操作上更简单，因此许多固定分区数据库选择不实施分区分割。因此，一开始配置的分区数就是您可以拥有的最大节点数

量，所以您需要选择足够多的分区以适应未来的增长。但是，每个分区也有管理开销，所以选择太大的数字会适得其反。

如果数据集的总大小难以预估（例如，如果它开始很小，但随着时间的推移可能会变得更大），选择正确的分区数是困难的。由于每个分区包含了总数据量固定比率的数据，因此每个分区的大小与集群中的数据总量成比例增长。如果分区非常大，再平衡和从节点故障恢复变得昂贵。但是，如果分区太小，则会产生太多的开销。当分区大小“恰到好处”的时候才能获得很好的性能，如果分区数量固定，但数据量变动很大，则难以达到最佳性能。

动态分区

对于使用键范围分区的数据库（参阅“按键范围分区”），具有固定边界的固定数量的分区将非常不便：如果出现边界错误，则可能会导致一个分区中的所有数据或者其他分区中的所有数据为空。手动重新配置分区边界将非常繁琐。

出于这个原因，按键的范围进行分区的数据库（如HBase和RethinkDB）会动态创建分区。当分区增长到超过配置的大小时（在HBase上，默认值是10GB），会被分成两个分区，每个分区约占一半的数据【26】。与之相反，如果大量数据被删除并且分区缩小到某个阈值以下，则可以将其与相邻分区合并。此过程与B树顶层发生的过程类似（参阅“B树”）。

每个分区分配给一个节点，每个节点可以处理多个分区，就像固定数量的分区一样。大型分区拆分后，可以将其中的一半转移到另一个节点，以平衡负载。在HBase中，分区文件的传输通过HDFS（底层分布式文件系统）来实现【3】。

动态分区的一个优点是分区数量适应总数据量。如果只有少量的数据，少量的分区就足够了，所以开销很小;如果有大量的数据，每个分区的大小被限制在一个可配置的最大值【23】。

需要注意的是，一个空的数据库从一个分区开始，因为没有关于在哪里绘制分区边界的先验信息。数据集开始时很小，直到达到第一个分区的分割点，所有写入操作都必须由单个节点处理，而其他节点则处于空闲状态。为了解决这个问题，HBase和MongoDB允许在一个空的数据库上配置一组初始分区（这被称为预分割（**pre-splitting**））。在键范围分区的情况中，预分割需要提前知道键是如何进行分配的【4,26】。

动态分区不仅适用于数据的范围分区，而且也适用于散列分区。从版本2.4开始，MongoDB同时支持范围和哈希分区，并且都是进行动态分割分区。

按节点比例分区

通过动态分区，分区的数量与数据集的大小成正比，因为拆分和合并过程将每个分区的大小保持在固定的最小值和最大值之间。另一方面，对于固定数量的分区，每个分区的大小与数据集的大小成正比。在这两种情况下，分区的数量都与节点的数量无关。

Cassandra和Ketama使用的第三种方法是使分区数与节点数成正比——换句话说，每个节点具有固定数量的分区【23,27,28】。在这种情况下，每个分区的大小与数据集大小成比例地增长，而节点数量保持不变，但是当增加节点数时，分区将再次变小。由于较大的数据量通常需要较大数量的节点进行存储，因此这种方法也使每个分区的大小较为稳定。

当一个新节点加入集群时，它随机选择固定数量的现有分区进行拆分，然后占有这些拆分分区中每个分区的一半，同时将每个分区的另一半留在原地。随机化可能会产生不公平的分割，但是平均在更大数量的分区上时（在Cassandra中，默认情况下，每个节点有256个分区），新节点最终从现有节点获得公平的负载份额。 Cassandra 3.0引入了另一种再分配的算法来避免不公平的分割【29】。

随机选择分区边界要求使用基于散列的分区（可以从散列函数产生的数字范围中挑选边界）。实际上，这种方法最符合一致性哈希的原始定义【7】（参阅“一致性哈希”）。最新的哈希函数可以在较低元数据开销的情况下达到类似的效果【8】。

运维：手动还是自动平衡

关于再平衡有一个重要问题：自动还是手动进行？

在全自动重新平衡（系统自动决定何时将分区从一个节点移动到另一个节点，无须人工干预）和完全手动（分区指派给节点由管理员明确配置，仅在管理员明确重新配置时才会更改）之间有一个权衡。例如，Couchbase，Riak和Voldemort会自动生成建议的分区分配，但需要管理员提交才能生效。

全自动重新平衡可以很方便，因为正常维护的操作工作较少。但是，这可能是不可预测的。再平衡是一个昂贵的操作，因为它需要重新路由请求并将大量数据从一个节点移动到另一个节点。如果没有做好，这个过程可能会使网络或节点负载过重，降低其他请求的性能。

这种自动化与自动故障检测相结合可能十分危险。例如，假设一个节点过载，并且对请求的响应暂时很慢。其他节点得出结论：过载的节点已经死亡，并自动重新平衡集群，使负载离开它。这会对已经超负荷的节点，其他节点和网络造成额外的负载，从而使情况变得更糟，并可能导致级联失败。

出于这个原因，再平衡的过程中有人参与是一件好事。这比完全自动的过程慢，但可以帮助防止运维意外。

请求路由

现在我们已经将数据集分割到多个机器上运行的多个节点上。但是仍然存在一个悬而未决的问题：当客户想要发出请求时，如何知道要连接哪个节点？随着分区重新平衡，分区对节点的分配也发生变化。为了回答这个问题，需要有人知晓这些变化：如果我想读或写键“foo”，需要连接哪个IP地址和端口号？

这个问题可以概括为 服务发现(**service discovery**) ，它不仅限于数据库。任何可通过网络访问的软件都有这个问题，特别是如果它的目标是高可用性（在多台机器上运行冗余配置）。许多公司已经编写了自己的内部服务发现工具，其中许多已经作为开源发布【30】。

概括来说，这个问题有几种不同的方案（如图6-7所示）：

1. 允许客户联系任何节点（例如，通过循环策略的负载均衡（**Round-Robin Load Balancer**））。如果该节点恰巧拥有请求的分区，则它可以直接处理该请求;否则，它将请求转发到适当的节点，接收回复并传递给客户端。
2. 首先将所有来自客户端的请求发送到路由层，它决定了应该处理请求的节点，并相应地转发。此路由层本身不处理任何请求；它仅负责分区的负载均衡。
3. 要求客户端知道分区和节点的分配。在这种情况下，客户端可以直接连接到适当的节点，而不需要任何中介。

以上所有情况中的关键问题是：作出路由决策的组件（可能是节点之一，还是路由层或客户端）如何了解分区-节点之间的分配关系变化？

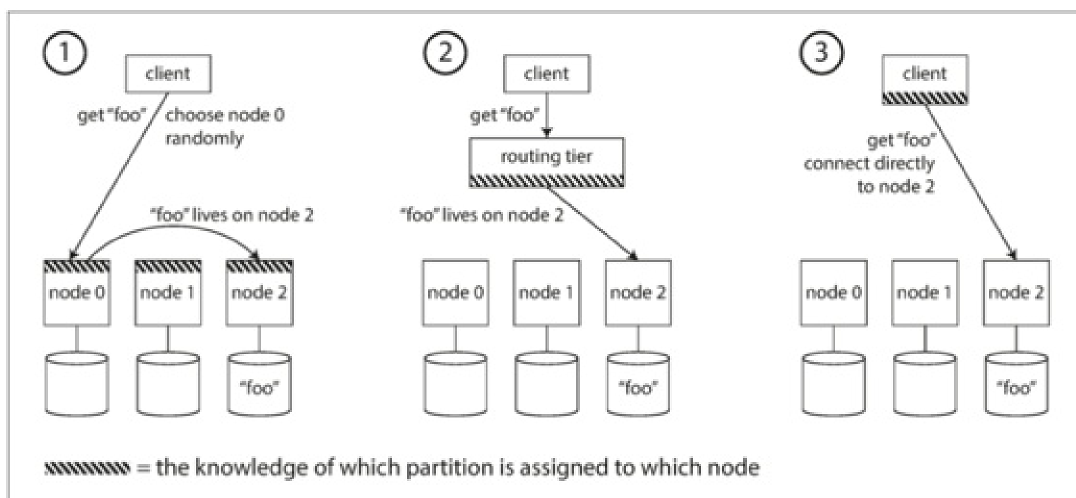


图**6-7** 将请求路由到正确节点的三种不同方式。

这是一个具有挑战性的问题，因为重要的是所有参与者都同意 - 否则请求将被发送到错误的节点，而不是正确处理。 在分布式系统中有达成共识的协议，但很难正确地实现（见第9章）。

许多分布式数据系统都依赖于一个独立的协调服务，比如ZooKeeper来跟踪集群元数据，如图6-8所示。 每个节点在ZooKeeper中注册自己，ZooKeeper维护分区到节点的可靠映射。 其他参与者（如路由层或分区感知客户端）可以在ZooKeeper中订阅此信息。 只要分区分配发生的改变，或者集群中添加或删除了一个节点，ZooKeeper就会通知路由层使路由信息保持最新状态。

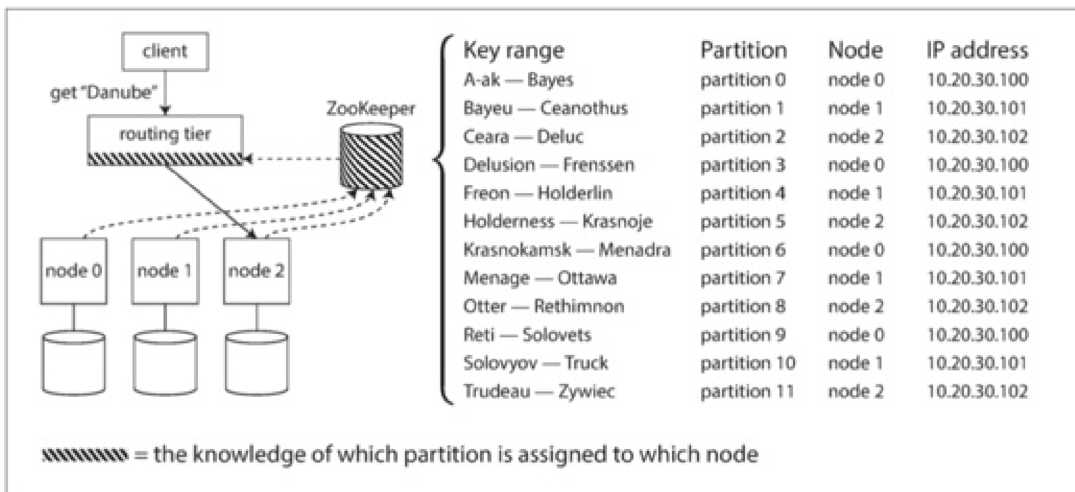


图**6-8** 使用**ZooKeeper**跟踪分区分配给节点。

例如，LinkedIn的Espresso使用Helix 【31】进行集群管理（依靠ZooKeeper），实现了如图6-8所示的路由层。 HBase，SolrCloud和Kafka也使用ZooKeeper来跟踪分区分配。MongoDB具有类似的体系结构，但它依赖于自己的配置服务器（**config server**） 实现和mongos守护进程作为路由层。

Cassandra和Riak采取不同的方法：他们在节点之间使用流言协议（**gossip protocol**） 来传播群集状态的变化。请求可以发送到任意节点，该节点会转发到包含所请求的分区的适当节点（图6-7中的方法1）。这个模型在数据库节点中增加了更多的复杂性，但是避免了对像ZooKeeper这样的外部协调服务的依赖。

Couchbase不会自动重新平衡，这简化了设计。通常情况下，它配置了一个名为moxi的路由层，它会从集群节点了解路由变化【32】。

当使用路由层或向随机节点发送请求时，客户端仍然需要找到要连接的IP地址。这些地址并不像分区的节点分布变化的那么快，所以使用DNS通常就足够了。

执行并行查询

到目前为止，我们只关注读取或写入单个键的非常简单的查询（对于文档分区的二级索引，另外还有分散/聚集查询）。这与大多数NoSQL分布式数据存储所支持的访问级别有关。

然而，通常用于分析的大规模并行处理（**MPP, Massively parallel processing**） 关系型数据库产品在其支持的查询类型方面要复杂得多。一个典型的数据仓库查询包含多个连接，过滤，分组和聚合操作。 MPP查询优化器将这个复杂的查询分解成许多执行阶段和分区，其中许多可以在数据库集群的不同节点上并行执行。涉及扫描大规模数据集的查询特别受益于这种并行执行。

数据仓库查询的快速并行执行是一个专门的话题，由于分析有很重要的商业意义，可以带来很多利益。我们将在第10章讨论并行查询执行的一些技巧。有关并行数据库中使用的技术的更详细的概述，请参阅参考文献【1,33】。

本章小结

在本章中，我们探讨了将大数据集划分成更小的子集的不同方法。数据量非常大的时候，在单台机器上存储和处理不再可行，则分区十分必要。分区的目标是在多台机器上均匀分布数据和查询负载，避免出现热点（负载不成比例的节点）。这需要选择适合于您的数据的分区方案，并在将节点添加到集群或从集群删除时进行再分区。

我们讨论了两种主要的分区方法：

键范围分区

其中键是有序的，并且分区拥有从某个最小值到某个最大值的所有键。排序的优势在于可以进行有效的范围查询，但是如果应用程序经常访问相邻的主键，则存在热点的风险。

在这种方法中，当分区变得太大时，通常将分区分成两个子分区，动态地再平衡分区。

散列分区

散列函数应用于每个键，分区拥有一定范围的散列。这种方法破坏了键的排序，使得范围查询效率低下，但可以更均匀地分配负载。

通过散列进行分区时，通常先提前创建固定数量的分区，为每个节点分配多个分区，并在添加或删除节点时将整个分区从一个节点移动到另一个节点。也可以使用动态分区。

两种方法搭配使用也是可行的，例如使用复合主键：使用键的一部分来标识分区，而使用另一部分作为排序顺序。

我们还讨论了分区和二级索引之间的相互作用。次级索引也需要分区，有两种方法：

- 按文档分区（本地索引），其中二级索引存储在与主键和值相同的分区中。这意味着只有一个分区需要在写入时更新，但是读取二级索引需要在所有分区之间进行分散/收集。
- 按关键词分区（全局索引），其中二级索引存在不同的分区的。辅助索引中的条目可以包括来自主键的所有分区的记录。当文档写入时，需要更新多个分区中的二级索引；但是可以从单个分区中进行读取。

最后，我们讨论了将查询路由到适当的分区的技术，从简单的分区负载平衡到复杂的并行查询执行引擎。

按照设计，多数情况下每个分区是独立运行的 — 这就是分区数据库可以扩展到多台机器的原因。但是，需要写入多个分区的操作结果可能难以预料：例如，如果写入一个分区成功，但另一个分区失败，会发生什么情况？我们将在下面的章节中讨论这个问题。

参考文献

1. David J. DeWitt and Jim N. Gray: "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992. doi:10.1145/129888.129894
2. Lars George: "HBase vs. BigTable Comparison," *larsgeorge.com*, November 2009.
3. "The Apache HBase Reference Guide," Apache Software Foundation, *hbase.apache.org*, 2014.
4. MongoDB, Inc.: "New Hash-Based Sharding Feature in MongoDB 2.4," *blog.mongodb.org*, April 10, 2013.
5. Ikai Lan: "App Engine Datastore Tip: Monotonically Increasing Values Are Bad," *ikaisays.com*, January 25, 2011.
6. Martin Kleppmann: "Java's hashCode Is Not Safe for Distributed Systems," *martin.kleppmann.com*, June 18, 2012.
7. David Karger, Eric Lehman, Tom Leighton, et al.: "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," at *29th Annual ACM Symposium on Theory of Computing* (STOC), pages 654–663, 1997. doi:10.1145/258533.258660
8. John Lamping and Eric Veach: "A Fast, Minimal Memory, Consistent Hash Algorithm," *arxiv.org*, June 2014.
9. Eric Redmond: "A Little Riak Book," Version 1.4.0, Basho Technologies, September 2013.
10. "Couchbase 2.5 Administrator Guide," Couchbase, Inc., 2014.
11. Avinash Lakshman and Prashant Malik: "Cassandra – A Decentralized Structured Storage System," at *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (LADIS), October 2009.
12. Jonathan Ellis: "Facebook's Cassandra Paper, Annotated and Compared to Apache Cassandra 2.0," *datastax.com*, September 12, 2013.
13. "Introduction to Cassandra Query Language," DataStax, Inc., 2014.
14. Samuel Axon: "3% of Twitter's Servers Dedicated to Justin Bieber," *mashable.com*, September 7, 2010.
15. "Riak 1.4.8 Docs," Basho Technologies, Inc., 2014.
16. Richard Low: "The Sweet Spot for Cassandra Secondary Indexing," *wentnet.com*, October 21, 2013.

17. Zachary Tong: “Customizing Your Document Routing,” *elasticsearch.org*, June 3, 2013.
18. “Apache Solr Reference Guide,” Apache Software Foundation, 2014.
19. Andrew Pavlo: “H-Store Frequently Asked Questions,” *hstore.cs.brown.edu*, October 2013.
20. “Amazon DynamoDB Developer Guide,” Amazon Web Services, Inc., 2014.
21. Rusty Klophaus: “Difference Between 2I and Search,” email to *riak-users* mailing list, *lists.basho.com*, October 25, 2011.
22. Donald K. Burleson: “Object Partitioning in Oracle,”*dba-oracle.com*, November 8, 2000.
23. Eric Evans: “Rethinking Topology in Cassandra,” at *ApacheCon Europe*, November 2012.
24. Rafał Kuć: “Reroute API Explained,” *elasticsearchserverbook.com*, September 30, 2013.
25. “Project Voldemort Documentation,” *project-voldemort.com*.
26. Enis Soztutar: “Apache HBase Region Splitting and Merging,” *hortonworks.com*, February 1, 2013.
27. Brandon Williams: “Virtual Nodes in Cassandra 1.2,” *datastax.com*, December 4, 2012.
28. Richard Jones: “libketama: Consistent Hashing Library for Memcached Clients,” *metabrew.com*, April 10, 2007.
29. Branimir Lambov: “New Token Allocation Algorithm in Cassandra 3.0,” *datastax.com*, January 28, 2016.
30. Jason Wilder: “Open-Source Service Discovery,” *jasonwilder.com*, February 2014.
31. Kishore Gopalakrishna, Shi Lu, Zhen Zhang, et al.: “Untangling Cluster Management with Helix,” at *ACM Symposium on Cloud Computing* (SoCC), October 2012. doi:10.1145/2391229.2391248
32. “Moxi 1.8 Manual,” Couchbase, Inc., 2014.
33. Shivnath Babu and Herodotos Herodotou: “Massively Parallel Databases and MapReduce Systems,” *Foundations and Trends in Databases*, volume 5, number 1, pages 1–104, November 2013.doi:10.1561/1900000036

上一章	目录	下一章
第五章：复制	设计数据密集型应用	第七章：事务

ou

7. 事务



> 一些作者声称，支持通用的两阶段提交代价太大，会带来性能与可用性的问题。让程序员来处理过度使用事务导致的性能问题，总比缺少事务编程好得多。
>
> ——James Corbett等人，Spanner：Google的全球分布式数据库（2012）

[TOC]

在数据系统的残酷现实中，很多事情都可能出错：

- 数据库软件、硬件可能在任意时刻发生故障（包括写操作进行到一半时）。
- 应用程序可能在任意时刻崩溃（包括一系列操作的中间）。
- 网络中断可能会意外切断数据库与应用的连接，或数据库之间的连接。
- 多个客户端可能会同时写入数据库，覆盖彼此的更改。
- 客户端可能读取到无意义的数据，因为数据只更新了一部分。
- 客户之间的竞争条件可能导致令人惊讶的错误。

为了实现可靠性，系统必须处理这些故障，确保它们不会导致整个系统的灾难性故障。但是实现容错机制工作量巨大。需要仔细考虑所有可能出错的事情，并进行大量的测试，以确保解决方案真正管用。

数十年来，事务（**transaction**）一直是简化这些问题的首选机制。事务是应用程序将多个读写操作组合成一个逻辑单元的一种方式。从概念上讲，事务中的所有读写操作被视作单个操作来执行：整个事务要么成功（提交（**commit**））要么失败（中止（**abort**），回滚（**rollback**））。如果失败，应用程序可以安全地重试。对于事务来说，应用程序的错误处理变得简单多了，因为它不用再担心部分失败的情况了，即某些操作成功，某些失败（无论出于何种原因）。

和事务打交道时间长了，你可能会觉得它显而易见。但我们不应将其视为理所当然。事务不自然法；它们是为了简化应用编程模型而创建的。通过使用事务，应用程序可以自由地忽略某些潜在的错误情况和并发问题，因为数据库会替应用处理好这些。（我们称之为安全保证（**safety guarantees**））。

并不是所有的应用都需要事务，有时候弱化事务保证、或完全放弃事务也是有好处的（例如，为了获得更高性能或更高可用性）。一些安全属性也可以在没有事务的情况下实现。

怎样知道你是否需要事务？为了回答这个问题，首先需要确切理解事务可以提供的安全保障，以及它们的代价。尽管乍看事务似乎很简单，但实际上有许多微妙但重要的细节在起作用。

本章将研究许多出错案例，并探索数据库用于防范这些问题的算法。尤其会深入并发控制的领域，讨论各种可能发生的竞争条件，以及数据库如何实现读已提交，快照隔离和可串行化等隔离级别。

本章同时适用于单机数据库与分布式数据库；在第8章中将重点讨论仅出现在分布式系统中的特殊挑战。

事务的棘手概念

现今，几乎所有的关系型数据库和一些非关系数据库都支持事务。其中大多数遵循IBM System R（第一个SQL数据库）在1975年引入的风格【1,2,3】。40年里，尽管一些实现细节发生了变化，但总体思路大同小异：MySQL，PostgreSQL，Oracle，SQL Server等数据库中的事务支持与System R异乎寻常地相似。

2000年以后，非关系（NoSQL）数据库开始普及。它们的目标是通过提供新的数据模型选择（参见第2章），并通过默认包含复制（第5章）和分区（第6章）来改善关系现状。事务是这种运动的主要原因：这些新一代数据库中的许多数据库完全放弃了事务，或者重新定义了这个词，描述比以前理解所更弱的一套保证【4】。

随着这种新型分布式数据库的炒作，人们普遍认为事务是可扩展性的对立面，任何大型系统都必须放弃事务以保持良好的性能和高可用性【5,6】。另一方面，数据库厂商有时将事务保证作为“重要应用”和“有价值数据”的基本要求。这两种观点都是纯粹的夸张。

事实并非如此简单：与其他技术设计选择一样，事务有其优势和局限性。为了理解这些权衡，让我们了解事务所提供保证的细节——无论是在正常运行中还是在各种极端（但是现实存在）情况下。

ACID的含义

事务所提供的安全保证，通常由众所周知的首字母缩略词ACID来描述，ACID代表原子性（**Atomicity**），一致性（**Consistency**），隔离性（**Isolation**）和持久性（**Durability**）。它由TheoHärder和Andreas Reuter于1983年创建，旨在为数据库中的容错机制建立精确的术语。

但实际上，不同数据库的ACID实现并不相同。例如，我们将会看到，围绕着隔离性（**Isolation**）的含义有许多含糊不清【8】。高层次上的想法是合理的，但魔鬼隐藏在细节里。今天，当一个系统声称自己“符合ACID”时，实际上能期待的是什么保证并不清楚。不幸的是，ACID现在几乎已经变成了一个营销术语。

（不符合ACID标准的系统有时被称为BASE，它代表基本可用性（**Basically Available**），软状态（**Soft State**）和最终一致性（**Eventual consistency**）【9】，这比ACID的定义更加模糊，似乎BASE的唯一合理的定义是“不是ACID”，即它几乎可以代表任何你想要的东西。）

让我们深入了解原子性，一致性，隔离性和持久性的定义，这可以让我们提炼出事务的思想。

原子性（**Atomicity**）

一般来说，原子是指不能分解成小部分的东西。这个词在计算的不同分支中意味着相似但又微妙不同的东西。例如，在多线程编程中，如果一个线程执行一个原子操作，这意味着另一个线程无法看到该操作的一半结果。系统只能处于操作之前或操作之后的状态，而不是介于两者之间的状态。

相比之下，ACID的原子性并不是关于并发（**concurrent**）的。它并不是在描述如果几个进程试图同时访问相同的数据会发生什么情况，这种情况包含在缩写**I**中，即隔离性（**Isolation**）

ACID的原子性描述了，当客户想进行多次写入，但在一些写操作处理完之后出现故障的情况。例如进程崩溃，网络连接中断，磁盘变满或者某种完整性约束被违反。如果这些写操作被分组到一个原子事务中，并且该事务由于错误而不能完成（提交），则该事务将被中止，并且数据库必须丢弃或撤消该事务中迄今为止所做的任何写入。

如果没有原子性，在多处更改进行到一半时发生错误，很难知道哪些更改已经生效，哪些没有生效。该应用程序可以再试一次，但冒着进行两次相同变更的风险，可能会导致数据重复或错误的数据。原子性简化了这个问题：如果事务被中止（**abort**），应用程序可以确定它没有改变任何东西，所以可以安全地重试。

ACID原子性的定义特征是：能够在错误时中止事务，丢弃该事务进行的所有写入变更的能力。或许可中止性（**abortability**）是更好的术语，但本书将继续使用原子性，因为这是惯用词。

一致性（Consistency）

一致性这个词重载的很厉害：

- 在第5章中，我们讨论了副本一致性，以及异步复制系统中的最终一致性问题（参阅“复制延迟问题”）。
- 一致性散列（Consistency Hash）)是某些系统用于重新分区的一种分区方法。
- 在CAP定理中，一致性一词用于表示可线性化。
- 在ACID的上下文中，一致性是指数据库在应用程序的特定概念中处于“良好状态”。

很不幸，这一个词就至少有四种不同的含义。

ACID一致性的概念是，对数据的一组特定陈述必须始终成立。即不变量（**invariants**）。例如，在会计系统中，所有账户整体上必须借贷相抵。如果一个事务开始于一个满足这些不变量的有效数据库，且在事务处理期间的任何写入操作都保持这种有效性，那么可以确定，不变量总是满足的。

但是，一致性的这种概念取决于应用程序对不变量的观念，应用程序负责正确定义它的事务，并保持一致性。这并不是数据库可以保证的事情：如果你写入违反不变量的脏数据，数据库也无法阻止你。（一些特定类型的不变量可以由数据库检查，例如外键约束或唯一约束，但是一般来说，是应用程序来定义什么样的数据是有效的，什么样是无效的。—— 数据库只管存储。）

原子性，隔离性和持久性是数据库的属性，而一致性（在ACID意义上）是应用程序的属性。应用可能依赖数据库的原子性和隔离属性来实现一致性，但这并不仅取决于数据库。因此，字母C不属于ACID[i]。

---

i

> i. 乔·海勒斯坦（Joe Hellerstein）指出，在论Härder与Reuter的论文中，“ACID中的C”是被“扔进去凑缩写单词的”【7】，而且那时候大家都不怎么在乎一致性。↩

隔离性（**Isolation**）

大多数数据库都会同时被多个客户端访问。如果它们各自读写数据库的不同部分，这是没有问题的，但是如果它们访问相同的数据库记录，则可能会遇到并发问题（竞争条件（**race conditions**））。

图7-1是这类问题的一个简单例子。假设你有两个客户端同时在数据库中增长一个计数器。（假设数据库中没有自增操作）每个客户端需要读取计数器的当前值，加 1 ，再回写新值。图7-1 中，因为发生了两次增长，计数器应该从42增至44；但由于竞态条件，实际上只增至 43 。

ACID意义上的隔离性意味着，同时执行的事务是相互隔离的：它们不能相互冒犯。传统的数据库教科书将隔离性形式化为可序列化（**Serializability**），这意味着每个事务可以假装它是唯一在整个数据库上运行的事务。数据库确保当事务已经提交时，结果与它们按顺序运行（一个接一个）是一样的，尽管实际上它们可能是并发运行的【10】。

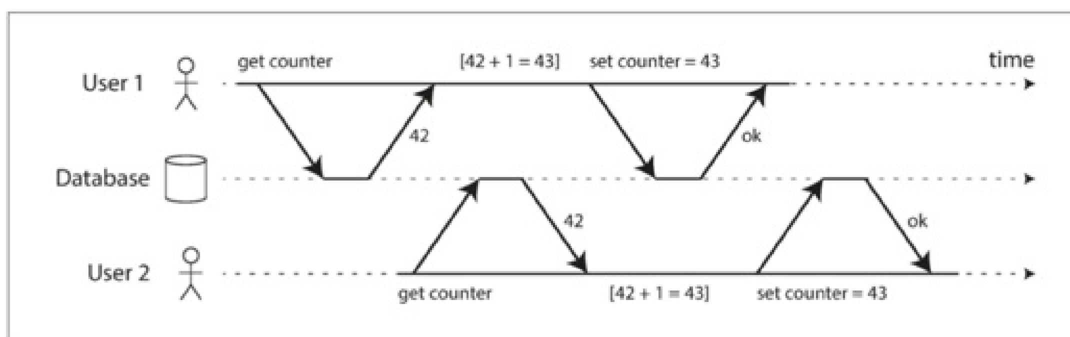


图**7-1** 两个客户之间的竞争状态同时递增计数器

然而实践中很少会使用可序列化隔离，因为它有性能损失。一些流行的数据库如Oracle 11g，甚至没有实现它。在Oracle中有一个名为“可序列化”的隔离级别，但实际上它实现了一种叫做快照隔离（**snapshot isolation**）的功能，这是一种比可序列化更弱的保证【8,11】。我们将在“弱隔离等级”中研究快照隔离和其他形式的隔离。

持久性（**Durability**）

数据库系统的目的是，提供一个安全的地方存储数据，而不用担心丢失。持久性是一个承诺，即一旦事务成功完成，即使发生硬件故障或数据库崩溃，写入的任何数据也不会丢失。

在单节点数据库中，持久性通常意味着数据已被写入非易失性存储设备，如硬盘或SSD。它通常还包括预写日志或类似的文件（参阅“让B树更可靠”），以便在磁盘上的数据结构损坏时进行恢复。在带复制的数据库中，持久性可能意味着数据已成功复制到一些节点。为了提供持久性保证，数据库必须等到这些写入或复制完成后，才能报告事务成功提交。

如“可靠性”一节所述，完美的持久性是不存在的：如果所有硬盘和所有备份同时被销毁，那显然没有任何数据库能救得了你。

> 复制和持久性
>
> 在历史上，持久性意味着写入归档磁带。后来它被理解为写入硬盘或SSD。最近它已经适应了“复制（replication）”的新内涵。哪种实现更好一些？
>
> 真相是，没有什么是完美的：
>
> - 如果你写入磁盘然后机器宕机，即使数据没有丢失，在修复机器或将磁盘转移到其他机器之前，也是无法访问的。这种情况下，复制系统可以保持可用性。
> - 一个相关性故障（停电，或一个特定输入导致所有节点崩溃的Bug）可能会一次性摧毁所有副本（参阅「可靠性」），任何仅存储在内存中的数据都会丢失，故内存数据库仍然要和磁盘写入打交道。
> - 在异步复制系统中，当主库不可用时，最近的写入操作可能会丢失（参阅「处理节点宕机」）。
> - 当电源突然断电时，特别是固态硬盘，有证据显示有时会违反应有的保证：甚至fsync也不能保证正常工作【12】。硬盘固件可能有错误，就像任何其他类型的软件一样【13,14】。
> - 存储引擎和文件系统之间的微妙交互可能会导致难以追踪的错误，并可能导致磁盘上的文件在崩溃后被损坏【15,16】。
> - 磁盘上的数据可能会在没有检测到的情况下逐渐损坏【17】。如果数据已损坏一段时间，副本和最近的备份也可能损坏。这种情况下，需要尝试从历史备份中恢复数据。
> - 一项关于固态硬盘的研究发现，在运行的前四年中，30％到80％的硬盘会产生至少一个坏块【18】。相比固态硬盘，磁盘的坏道率较低，但完全失效的概率更高。
> - 如果SSD断电，可能会在几周内开始丢失数据，具体取决于温度【19】。
>
> 在实践中，没有一种技术可以提供绝对保证。只有各种降低风险的技术，包括写入磁盘，复制到远程机器和备份——它们可以且应该一起使用。与往常一样，最好抱着怀疑的态度接受任何理论上的“保证”

单对象和多对象操作

回顾一下，在ACID中，原子性和隔离性描述了客户端在同一事务中执行多次写入时，数据库应该做的事情：

原子性

如果在一系列写操作的中途发生错误，则应中止事务处理，并丢弃当前事务的所有写入。换句话说，数据库免去了用户对部分失败的担忧——通过提供“宁为玉碎，不为瓦全（**all-or-nothing**）”的保证。

隔离性

同时运行的事务不应该互相干扰。例如，如果一个事务进行多次写入，则另一个事务要么看到全部写入结果，要么什么都看不到，但不应该是一些子集。

这些定义假设你想同时修改多个对象（行，文档，记录）。通常需要多对象事务（**multi-object transaction**）来保持多块数据同步。图7-2展示了一个来自电邮应用的例子。执行以下查询来显示用户未读邮件数量：

```
SELECT COUNT（*）FROM emails WHERE recipient_id = 2 AND unread_flag = true
```

但如果邮件太多，你可能会觉得这个查询太慢，并决定用单独的字段存储未读邮件的数量（一种反规范化）。现在每当一个新消息写入时，必须也增长未读计数器，每当一个消息被标记为已读时，也必须减少未读计数器。

在图7-2中，用户2 遇到异常情况：邮件列表里显示有未读消息，但计数器显示为零未读消息，因为计数器增长还没有发生[ii]。隔离性可以避免这个问题：通过确保用户2 要么同时看到新邮件和增长后的计数器，要么都看不到。反正不会看到执行到一半的中间结果。

> ii. 可以说邮件应用中的错误计数器并不是什么特别重要的问题。但换种方式来看，你可以把未读计数器换成客户账户余额，把邮件收发看成支付交易。 ↩

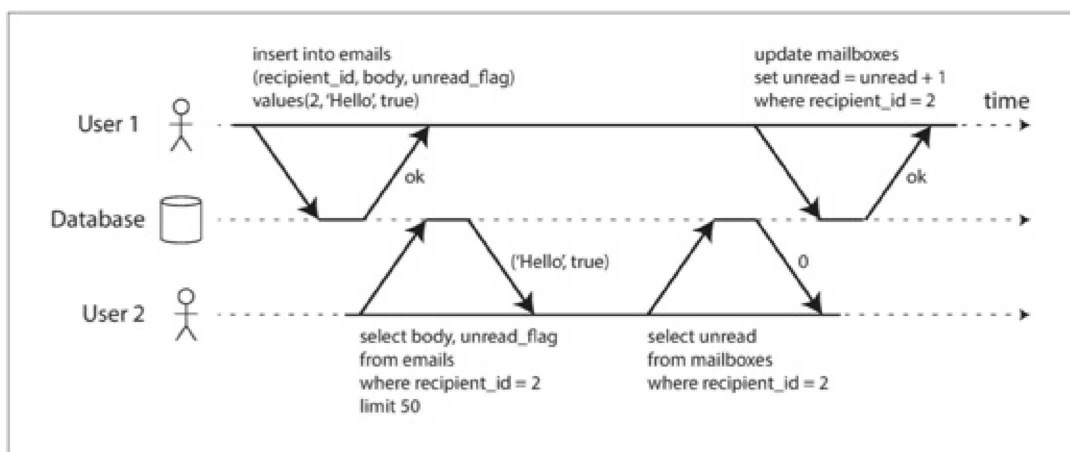


图**7-2** 违反隔离性：一个事务读取另一个事务的未被执行的写入（“脏读”）。

图7-3说明了对原子性的需求：如果在事务过程中发生错误，邮箱和未读计数器的内容可能会失去同步。在原子事务中，如果对计数器的更新失败，事务将被中止，并且插入的电子邮件将被回滚。

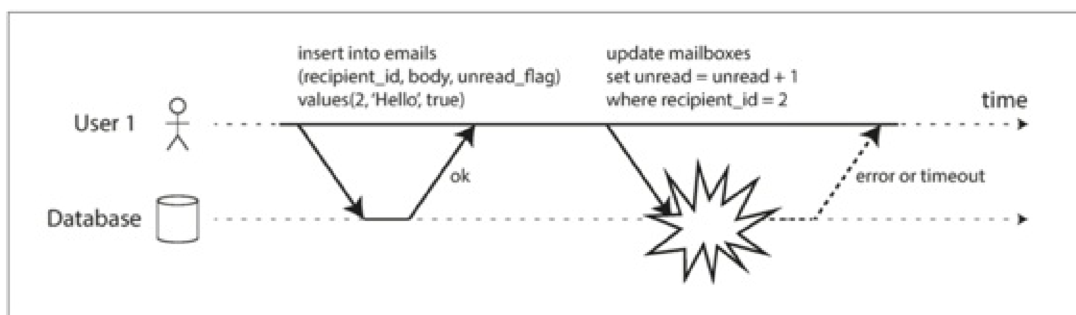


图**7-3** 原子性确保发生错误时，事务先前的任何写入都会被撤消，以避免状态不一致

多对象事务需要某种方式来确定哪些读写操作属于同一个事务。在关系型数据库中，通常基于客户端与数据库服务器的TCP连接：在任何特定连接上，`BEGIN TRANSACTION` 和 `COMMIT` 语句之间的所有内容，被认为是同一事务的一部分.[iii]

> iii. 这并不完美。如果TCP连接中断，则事务必须中止。如果中断发生在客户端请求提交之后，但在服务器确认提交发生之前，客户端并不知道事务是否已提交。为了解决这个问题，事务管理器可以通过一个唯一事务标识符来对操作进行分组，这个标识符并未绑定到特定TCP连接。后续再“数据库端到端的争论”一节将回到这个主题。↩

另一方面，许多非关系数据库并没有将这些操作组合在一起的方法。即使存在多对象API（例如，键值存储可能具有在一个操作中更新几个键的多重放置操作），但这并不一定意味着它具有事务语义：该命令可能在一些键上成功，在其他的键上失败，使数据库处于部分更新的状态。

单对象写入

当单个对象发生改变时，原子性和隔离也是适用的。例如，假设您正在向数据库写入一个 20 KB的 JSON文档：

- 如果在发送第一个10 KB之后网络连接中断，数据库是否存储了不可解析的10KB JSON片段？
- 如果在数据库正在覆盖磁盘上的前一个值的过程中电源发生故障，是否最终将新旧值拼接在一起？
- 如果另一个客户端在写入过程中读取该文档，是否会看到部分更新的值？

这些问题非常让人头大，故存储引擎一个几乎普遍的目标是：对单节点上的单个对象（例如键值对）上提供原子性和隔离性。原子性可以通过使用日志来实现崩溃恢复（参阅“使B树可靠”），并且可以使用每个对象上的锁来实现隔离（每次只允许一个线程访问对象））。

一些数据库也提供更复杂的原子操作，例如自增操作，这样就不再需要像 图7-1 那样的读取-修改-写入序列了。同样流行的是比较和设置（**CAS, compare-and-set**）操作，当值没有并发被其他人修改过时，才允许执行写操作。

这些单对象操作很有用，因为它们可以防止在多个客户端尝试同时写入同一个对象时丢失更新（参阅“防止丢失更新”）。但它们不是通常意义上的事务。CAS以及其他单一对象操作被称为“轻量级事务”，甚至出于营销目的被称为“ACID”【20,21,22】，但是这个术语是误导性的。事务通常被理解为，将多个对象上的多个操作合并为一个执行单元的机制。[iv]

> [iv]. 严格地说，原子自增（**atomic increment**）这个术语在多线程编程的意义上使用了原子这个词。 在ACID的情况下，它实际上应该被称为孤立（**isolated**）的或可序列化（**serializable**）的增量。 但这就太吹毛求疵了。 ↩

多对象事务的需求

许多分布式数据存储已经放弃了多对象事务，因为多对象事务很难跨分区实现，而且在需要高可用性或高性能的情况下，它们可能会碍事。但说到底，在分布式数据库中实现事务，并没有什么根本性的障碍。第9章 将讨论分布式事务的实现。

但是我们是否需要多对象事务？是否有可能只用键值数据模型和单对象操作来实现任何应用程序？

有一些场景中，单对象插入，更新和删除是足够的。但是许多其他场景需要协调写入几个不同的对象：

- 在关系数据模型中，一个表中的行通常具有对另一个表中的行的外键引用。 （类似的是，在一个图数据模型中，一个顶点有着到其他顶点的边）。多对象事务使你确信这些引用始终有效：当插入几个相互引用的记录时，外键必须是正确的，最新的，不然数据就没有意义。
- 在文档数据模型中，需要一起更新的字段通常在同一个文档中，这被视为单个对象——更新单个文档时不需要多对象事务。但是，缺乏连接功能的文档数据库会鼓励非规范化（参阅“关系型数据库与文档数据库在今日的对比”）。当需要更新非规范化的信息时，如图7-2 所示，需要一次更新多个文档。事务在这种情况下非常有用，可以防止非规范化的数据不同步。
- 在具有二级索引的数据库中（除了纯粹的键值存储以外几乎都有），每次更改值时都需要更新索引。从事务角度来看，这些索引是不同的数据库对象：例如，如果没有事务隔离性，记录可能出现在一个索引中，但没有出现在另一个索引中，因为第二个索引的更新还没有发生。

这些应用仍然可以在没有事务的情况下实现。然而，没有原子性，错误处理就要复杂得多，缺乏隔离性，就会导致并发问题。我们将在“弱隔离级别”中讨论这些问题，并在第12章中探讨其他方法。

处理错误和中止

事务的一个关键特性是，如果发生错误，它可以中止并安全地重试。 ACID数据库基于这样的哲学：如果数据库有违反其原子性，隔离性或持久性的危险，则宁愿完全放弃事务，而不是留下半成品。

然而并不是所有的系统都遵循这个哲学。特别是具有无主复制的数据存储，主要是在“尽力而为”的基础上进行工作。可以概括为“数据库将做尽可能多的事，运行遇到错误时，它不会撤消它已经完成的事情“ ——所以，从错误中恢复是应用程序的责任。

错误发生不可避免，但许多软件开发人员倾向于只考虑乐观情况，而不是错误处理的复杂性。例如，像Rails的ActiveRecord和Django这样的对象关系映射（**ORM, object-relation Mapping**）框架不会重试中断的事务—— 这个错误通常会导致一个从堆栈向上传播的异常，所以任何用户输入都会被丢弃，用户拿到一个错误信息。这实在是太耻辱了，因为中止的重点就是允许安全的重试。

尽管重试一个中止的事务是一个简单而有效的错误处理机制，但它并不完美：

- 如果事务实际上成功了，但是在服务器试图向客户端确认提交成功时网络发生故障（所以客户端认为提交失败了），那么重试事务会导致事务被执行两次——除非你有一个额外的应用级除重机制。
- 如果错误是由于负载过大造成的，则重试事务将使问题变得更糟，而不是更好。为了避免这种正反馈循环，可以限制重试次数，使用指数退避算法，并单独处理与过载相关的错误（如果允许）。
- 仅在临时性错误（例如，由于死锁，异常情况，临时性网络中断和故障转移）后才值得重试。在发生永久性错误（例如，违反约束）之后重试是毫无意义的。
- 如果事务在数据库之外也有副作用，即使事务被中止，也可能发生这些副作用。例如，如果你正在发送电子邮件，那你肯定不希望每次重试事务时都重新发送电子邮件。如果你想确保几个不同的系统一起提交或放弃，二阶段提交（**2PC, two-phase commit**）可以提供帮助（“原子提交和两阶段提交（2PC）”中将讨论这个问题）。
- 如果客户端进程在重试中失效，任何试图写入数据库的数据都将丢失。

弱隔离级别

如果两个事务不触及相同的数据，它们可以安全地并行（**parallel**）运行，因为两者都不依赖于另一个。当一个事务读取由另一个事务同时修改的数据时，或者当两个事务试图同时修改相同的数据时，并发问题（竞争条件）才会出现。

并发BUG很难通过测试找到，因为这样的错误只有在特殊时机下才会触发。这样的时机可能很少，通常很难重现[译注i]。并发性也很难推理，特别是在大型应用中，你不一定知道哪些其他代码正在访问数据库。在一次只有一个用户时，应用开发已经很麻烦了，有许多并发用户使得它更加困难，因为任何一个数据都可能随时改变。

> 译注i. 轶事：偶然出现的瞬时错误有时称为***Heisenbug***，而确定性的问题对应地称为***Bohrbugs*** ↩

出于这个原因，数据库一直试图通过提供事务隔离（**transaction isolation**）来隐藏应用程序开发者的并发问题。从理论上讲，隔离可以通过假装没有并发发生，让你的生活更加轻松：可序列化（**serializable**）的隔离等级意味着数据库保证事务的效果与连续运行（即一次一

个，没有任何并发）是一样的。

实际上不幸的是：隔离并没有那么简单。可序列化会有性能损失，许多数据库不愿意支付这个代价【8】。因此，系统通常使用较弱的隔离级别来防止一部分，而不是全部的并发问题。这些隔离级别难以理解，并且会导致微妙的错误，但是它们仍然在实践中被使用【23】。

并发性错误导致的并发性错误不仅仅是一个理论问题。它们造成了很多的资金损失【24,25】，耗费了财务审计人员的调查【26】，并导致客户数据被破坏【27】。关于这类问题的一个流行的评论是“如果你正在处理财务数据，请使用ACID数据库！”—— 但是这一点没有提到。即使是很多流行的关系型数据库系统（通常被认为是“ACID”）也使用弱隔离级别，所以它们也不一定能防止这些错误的发生。

比起盲目地依赖工具，我们应该对存在的并发问题的种类，以及如何防止这些问题有深入的理解。然后就可以使用我们所掌握的工具来构建可靠和正确的应用程序。

在本节中，我们将看几个在实践中使用的弱（不可串行化（**nonserializable**））隔离级别，并详细讨论哪种竞争条件可能发生也可能不发生，以便您可以决定什么级别适合您的应用程序。一旦我们完成了这个工作，我们将详细讨论可串行性（请参阅“可序列化”）。我们讨论的隔离级别将是非正式的，使用示例。如果你需要严格的定义和分析它们的属性，你可以在学术文献中找到它们[28,29,30]。

读已提交

最基本的事务隔离级别是读已提交（**Read Committed**）[V]，它提供了两个保证：

1. 从数据库读时，只能看到已提交的数据（没有脏读（**dirty reads**））。
2. 写入数据库时，只会覆盖已经写入的数据（没有脏写（**dirty writes**））。

我们来更详细地讨论这两个保证。

> V. 某些数据库支持甚至更弱的隔离级别，称为读未提交（**Read uncommitted**）。它可以防止脏写，但不防止脏读。↩

没有脏读

设想一个事务已经将一些数据写入数据库，但事务还没有提交或中止。另一个事务可以看到未提交的数据吗？如果是的话，那就叫做脏读（**dirty reads**）【2】。

在读已提交隔离级别运行的事务必须防止脏读。这意味着事务的任何写入操作只有在该事务提交时才能被其他人看到（然后所有的写入操作都会立即变得可见）。如图7-4所示，用户1设置了 `x = 3` ，但用户2 的 `get x` 仍旧返回旧值2 ，而用户1 尚未提交。

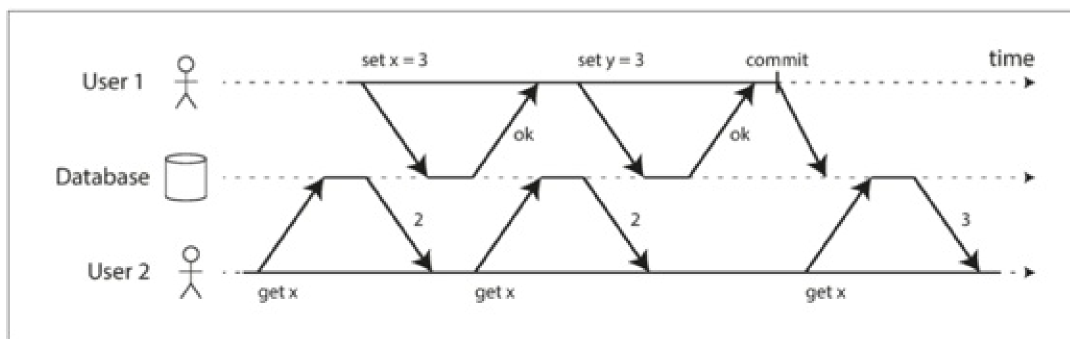


图**7-4** 没有脏读：用户**2**只有在用户**1**的事务已经提交后才能看到**x**的新值。

为什么要防止脏读，有几个原因：

- 如果事务需要更新多个对象，脏读取意味着另一个事务可能会只看到一部分更新。例如，在图7-2中，用户看到新的未读电子邮件，但看不到更新的计数器。这就是电子邮件的脏读。看到处于部分更新状态的数据库会让用户感到困惑，并可能导致其他事务做出错误的决定。
- 如果事务中止，则所有写入操作都需要回滚（如图7-3所示）。如果数据库允许脏读，那就意味着一个事务可能会看到稍后需要回滚的数据，即从未实际提交给数据库的数据。想想后果就让人头大。

没有脏写

如果两个事务同时尝试更新数据库中的相同对象，会发生什么情况？我们不知道写入的顺序是怎样的，但是我们通常认为后面的写入会覆盖前面的写入。

但是，如果先前的写入是尚未提交事务的一部分，又会发生什么情况，后面的写入会覆盖一个尚未提交的值？这被称作脏写（**dirty write**）【28】。在读已提交的隔离级别上运行的事务必须防止脏写，通常是延迟第二次写入，直到第一次写入事务提交或中止为止。

通过防止脏写，这个隔离级别避免了一些并发问题：

- 如果事务更新多个对象，脏写会导致不好的结果。例如，考虑 图7-5，图7-5 以一个二手车销售网站为例，Alice和Bob两个人同时试图购买同一辆车。购买汽车需要两次数据库写入：网站上的商品列表需要更新，以反映买家的购买，销售发票需要发送给买家。在图7-5的情况下，销售是属于Bob的（因为他成功更新了商品列表），但发票却寄送给了爱丽丝（因为她成功更新了发票表）。读已提交会阻止这样这样的事故。
- 但是，提交读取并不能防止图7-1中两个计数器增量之间的竞争状态。在这种情况下，第二次写入发生在第一个事务提交后，所以它不是一个脏写。这仍然是不正确的，但是出于不同的原因，在“防止更新丢失”中将讨论如何使这种计数器增量安全。

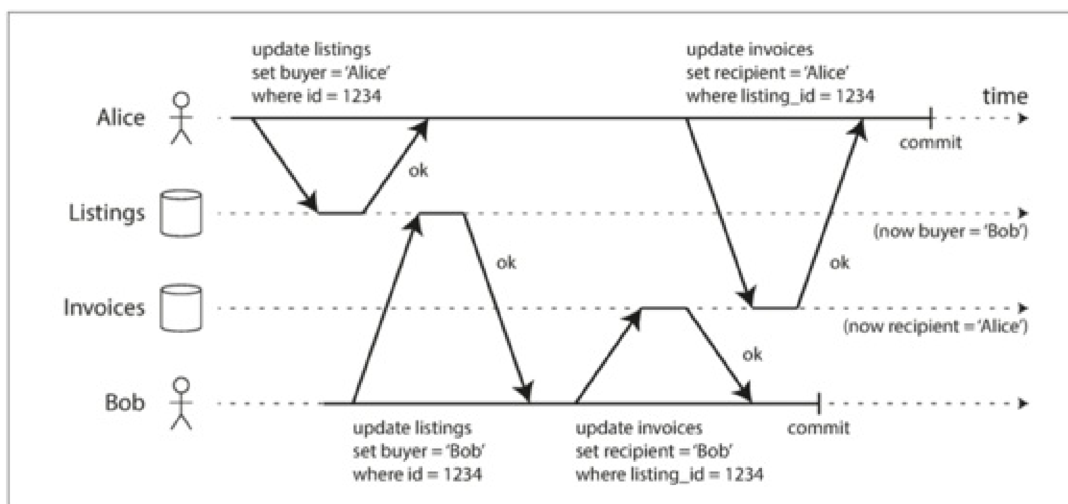


图**7-5** 如果存在脏写，来自不同事务的冲突写入可能会混淆在一起

实现读已提交

读已提交是一个非常流行的隔离级别。这是Oracle 11g，PostgreSQL，SQL Server 2012，MemSQL和其他许多数据库的默认设置【8】。

最常见的情况是，数据库通过使用行锁（**row-level lock**）来防止脏写：当事务想要修改特定对象（行或文档）时，它必须首先获得该对象的锁。然后必须持有该锁直到事务被提交或中止。一次只有一个事务可持有任何给定对象的锁；如果另一个事务要写入同一个对象，则必须等到第一个事务提交或中止后，才能获取该锁并继续。这种锁定是读已提交模式（或更强的隔离级别）的数据库自动完成的。

如何防止脏读？一种选择是使用相同的锁，并要求任何想要读取对象的事务来简单地获取该锁，然后在读取之后立即再次释放该锁。这能确保不会读取进行时，对象不会在脏的状态，有未提交的值（因为在那段时间锁会被写入该对象的事务持有）。

但是要求读锁的办法在实践中效果并不好。因为一个长时间运行的写入事务会迫使许多只读事务等到这个慢写入事务完成。这会损失只读事务的响应时间，并且不利于可操作性：因为等待锁，应用某个部分的迟缓可能由于连锁效应，导致其他部分出现问题。

出于这个原因，大多数数据库[vi]使用图7-4的方式防止脏读：对于写入的每个对象，数据库都会记住旧的已提交值，和由当前持有写入锁的事务设置的新值。 当事务正在进行时，任何其他读取对象的事务都会拿到旧值。 只有当新值提交后，事务才会切换到读取新值。

> vi. 在撰写本文时，唯一在读已提交隔离级别使用读锁的主流数据库是使用`read_committed_snapshot = off`配置的IBM DB2和Microsoft SQL Server [23,36]。↩

快照隔离和可重复读

如果只从表面上看读已提交隔离级别你就认为它完成了事务所需的一切，那是可以原谅的。它允许中止（原子性的要求）；它防止读取不完整的事务结果，并排写入的并发写入。事实上这些功能非常有用，比起没有事务的系统来，可以提供更多的保证。

但是在使用此隔离级别时，仍然有很多地方可能会产生并发错误。例如图7-6说明了读已提交时可能发生的问题。

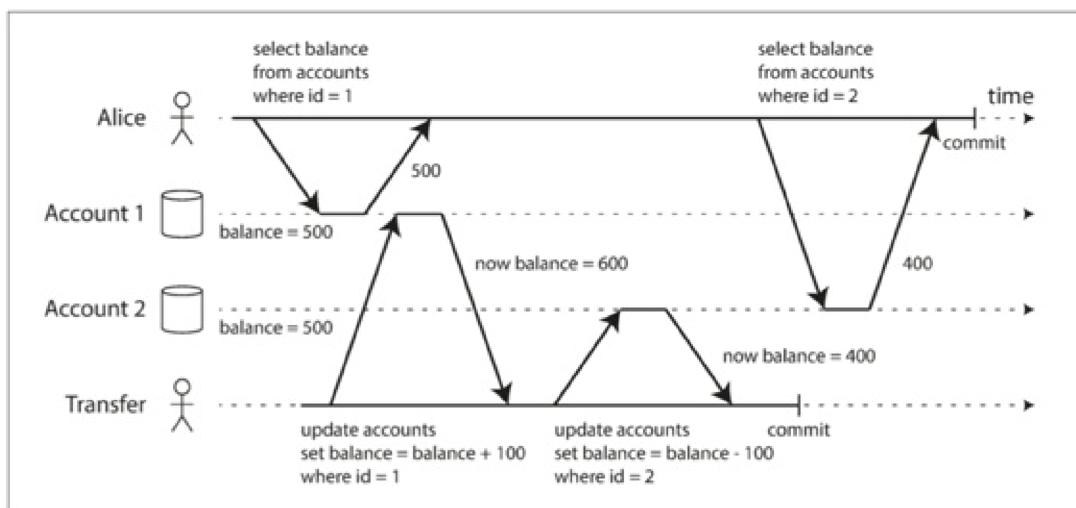


图**7-6** 读取偏差：**Alice**观察数据库处于不一致的状态

爱丽丝在银行有1000美元的储蓄，分为两个账户，每个500美元。现在一笔事务从她的一个账户中转移了100美元到另一个账户。如果她在事务处理的同时查看其账户余额列表，不幸地在转账事务完成前看到收款账户余额（余额为500美元），而在转账完成后看到另一个转出账户（已经转出100美元，余额400美元）。对爱丽丝来说，现在她的账户似乎只有900美元——看起来100美元已经消失了。

这种异常被称为不可重复读（**nonrepeatable read**）或读取偏差（**read skew**）：如果Alice在事务结束时再次读取账户1的余额，她将看到与她之前的查询中看到的不同的值（600美元）。在读已提交的隔离条件下，不可重复读被认为是可接受的：Alice看到的帐户余额时确实在阅读时已经提交了。

> 不幸的是，术语偏差（**skew**） 这个词是过载的：以前使用它是因为热点的不平衡工作量（参阅“偏斜的负载倾斜与消除热点”），而这里偏差意味着异常的时机。

对于Alice的情况，这不是一个长期持续的问题。因为如果她几秒钟后刷新银行网站的页面，她很可能会看到一致的帐户余额。但是有些情况下，不能容忍这种暂时的不一致：

备份

进行备份需要复制整个数据库，对大型数据库而言可能需要花费数小时才能完成。备份进程运行时，数据库仍然会接受写入操作。因此备份可能会包含一些旧的部分和一些新的部分。如果从这样的备份中恢复，那么不一致（如消失的钱）就会变成永久的。

分析查询和完整性检查

有时，您可能需要运行一个查询，扫描大部分的数据库。这样的查询在分析中很常见（参阅“事务处理或分析？”），也可能是定期完整性检查（即监视数据损坏）的一部分。如果这些查询在不同时间点观察数据库的不同部分，则可能会返回毫无意义的结果。

快照隔离（**snapshot isolation**）【28】是这个问题最常见的解决方案。想法是，每个事务都从数据库的一致快照（**consistent snapshot**）中读取——也就是说，事务可以看到事务开始时在数据库中提交的所有数据。即使这些数据随后被另一个事务更改，每个事务也只能看到该特定时间点的旧数据。

快照隔离对长时间运行的只读查询（如备份和分析）非常有用。如果查询的数据在查询执行的同时发生变化，则很难理解查询的含义。当一个事务可以看到数据库在某个特定时间点冻结时的一致快照，理解起来就很容易了。

快照隔离是一个流行的功能：PostgreSQL，使用InnoDB引擎的MySQL，Oracle，SQL Server等都支持【23,31,32】。

实现快照隔离

与读取提交的隔离类似，快照隔离的实现通常使用写锁来防止脏写（请参阅“读已提交”），这意味着进行写入的事务会阻止另一个事务修改同一个对象。但是读取不需要任何锁定。从性能的角度来看，快照隔离的一个关键原则是：读不阻塞写，写不阻塞读。这允许数据库在处理一致性快照上的长时间查询时，可以正常地同时处理写入操作。且两者间没有任何锁定争用。

为了实现快照隔离，数据库使用了我们看到的用于防止图7-4中的脏读的机制的一般化。数据库必须可能保留一个对象的几个不同的提交版本，因为各种正在进行的事务可能需要看到数据库在不同的时间点的状态。因为它并排维护着多个版本的对象，所以这种技术被称为多版本并发控制（**MVCC, multi-version concurrentcy control**）。

如果一个数据库只需要提供读已提交的隔离级别，而不提供快照隔离，那么保留一个对象的两个版本就足够了：提交的版本和被覆盖但尚未提交的版本。支持快照隔离的存储引擎通常也使用MVCC来实现读已提交隔离级别。一种典型的方法是读已提交为每个查询使用单独的快照，而快照隔离对整个事务使用相同的快照。

图7-7说明了如何在PostgreSQL中实现基于MVCC的快照隔离【31】（其他实现类似）。当一个事务开始时，它被赋予一个唯一的，永远增长[vii]的事务ID（`txid`）。每当事务向数据库写入任何内容时，它所写入的数据都会被标记上写入者的事务ID。

> vii. 事实上，事务ID是32位整数，所以大约会在40亿次事务之后溢出。 PostgreSQL的Vacuum过程会清理老旧的事务ID，确保事务ID溢出（回卷）不会影响到数据。 ↩

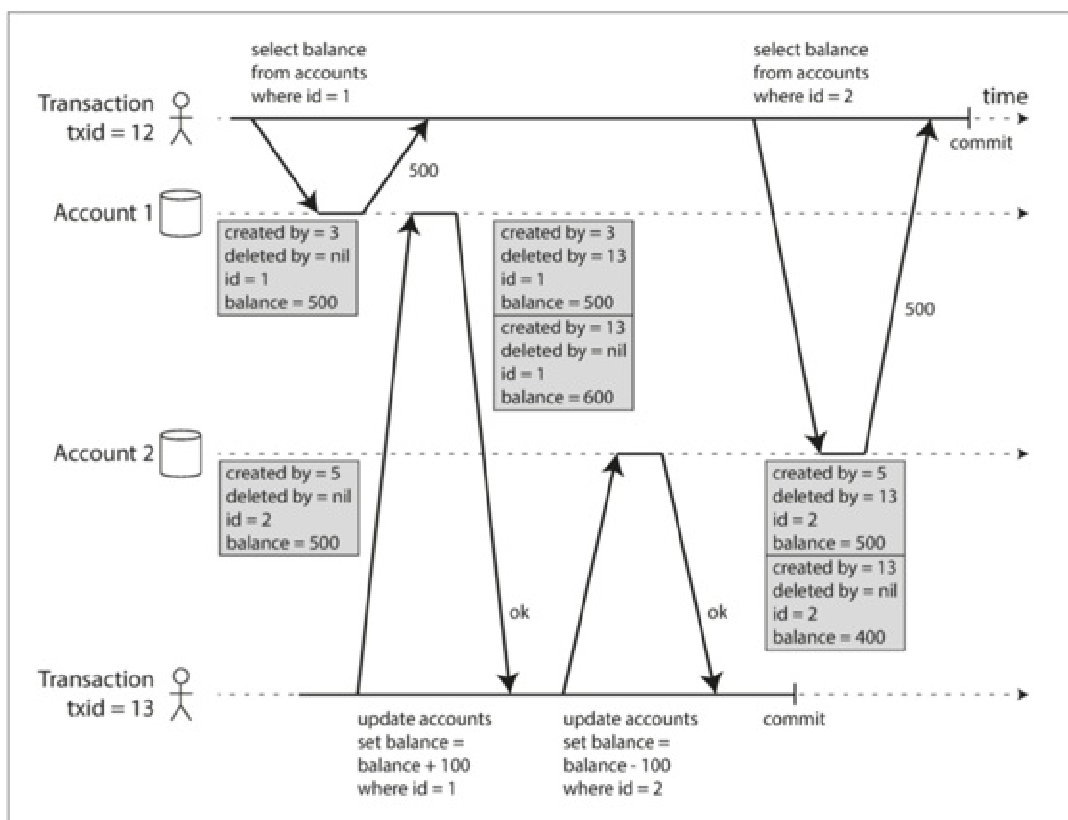


图7-7 使用多版本对象实现快照隔离

表中的每一行都有一个 `created_by` 字段，其中包含将该行插入到表中的的事务ID。此外，每行都有一个 `deleted_by` 字段，最初是空的。如果某个事务删除了一行，那么该行实际上并未从数据库中删除，而是通过将 `deleted_by` 字段设置为请求删除的事务的ID来标记为删除。在稍后的时间，当确定没有事务可以再访问已删除的数据时，数据库中的垃圾收集过程会将所有带有删除标记的行移除，并释放其空间。[译注ii]

> 译注ii. 在PostgreSQL中，`created_by` 的实际名称为 `xmin`，`deleted_by` 的实际名称为 `xmax` ↩

`UPDATE` 操作在内部翻译为 `DELETE` 和 `INSERT`。例如，在图7-7中，事务13 从账户2 中扣除100美元，将余额从500美元改为400美元。实际上包含两条账户2 的记录：余额为 $500 的行被标记为被事务**13**删除，余额为 $400 的行由事务**13**创建。

观察一致性快照的可见性规则

当一个事务从数据库中读取时，事务ID用于决定它可以看见哪些对象，看不见哪些对象。通过仔细定义可见性规则，数据库可以向应用程序呈现一致的数据库快照。工作如下：

1. 在每次事务开始时，数据库列出当时所有其他（尚未提交或中止）的事务清单，即使之后提交了，这些事务的写入也都会被忽略。
2. 被中止事务所执行的任何写入都将被忽略。
3. 由具有较晚事务ID（即，在当前事务开始之后开始的）的事务所做的任何写入都被忽略，而不管这些事务是否已经提交。

4. 所有其他写入，对应用都是可见的。

这些规则适用于创建和删除对象。在图7-7中，当事务12 从账户2 读取时，它会看到 $500 的余额，因为 $500 余额的删除是由事务13 完成的（根据规则3，事务12 看不到事务13 执行的删除），且400美元记录的创建也是不可见的（按照相同的规则）。

换句话说，如果以下两个条件都成立，则可见一个对象：

- 读事务开始时，创建该对象的事务已经提交。
- 对象未被标记为删除，或如果被标记为删除，请求删除的事务在读事务开始时尚未提交。

长时间运行的事务可能会长时间使用快照，并继续读取（从其他事务的角度来看）早已被覆盖或删除的值。由于从来不更新值，而是每次值改变时创建一个新的版本，数据库可以在提供一致快照的同时只产生很小的额外开销。

索引和快照隔离

索引如何在多版本数据库中工作？一种选择是使索引简单地指向对象的所有版本，并且需要索引查询来过滤掉当前事务不可见的任何对象版本。当垃圾收集删除任何事务不再可见的旧对象版本时，相应的索引条目也可以被删除。

在实践中，许多实现细节决定了多版本并发控制的性能。例如，如果同一对象的不同版本可以放入同一个页面中，PostgreSQL的优化可以避免更新索引【31】。

在CouchDB，Datomic和LMDB中使用另一种方法。虽然它们也使用B树，但它们使用的是一种仅追加/写时拷贝（**append-only/copy-on-write**）的变体，它们在更新时不覆盖树的页面，而为每个修改页面创建一份副本。从父页面直到树根都会级联更新，以指向它们子页面的新版本。任何不受写入影响的页面都不需要被复制，并且保持不变【33,34,35】。

使用仅追加的B树，每个写入事务（或一批事务）都会创建一颗新的B树，当创建时，从该特定树根生长的树就是数据库的一个一致性快照。没必要根据事务ID过滤掉对象，因为后续写入不能修改现有的B树；它们只能创建新的树根。但这种方法也需要一个负责压缩和垃圾收集的后台进程。

可重复读与命名混淆

快照隔离是一个有用的隔离级别，特别对于只读事务而言。但是，许多数据库实现了它，却用不同的名字来称呼。在Oracle中称为可序列化（**Serializable**）的，在PostgreSQL和MySQL中称为可重复读（**repeatable read**）【23】。

这种命名混淆的原因是SQL标准没有快照隔离的概念，因为标准是基于System R 1975年定义的隔离级别【2】，那时候快照隔离尚未发明。相反，它定义了可重复读，表面上看起来与快照隔离很相似。 PostgreSQL和MySQL称其快照隔离级别为可重复读（**repeatable read**），因为这样符合标准要求，所以它们可以声称自己“标准兼容”。

不幸的是，SQL标准对隔离级别的定义是有缺陷的——模糊，不精确，并不像标准应有的样子独立于实现【28】。有几个数据库实现了可重复读，但它们实际提供的保证存在很大的差异，尽管表面上是标准化的【23】。在研究文献【29,30】中已经有了可重复读的正式定义，但大多数的实现并不能满足这个正式定义。最后，IBM DB2使用“可重复读”来引用可串行化【8】。

结果，没有人真正知道可重复读的意思。

防止丢失更新

到目前为止已经讨论的读已提交和快照隔离级别，主要保证了只读事务在并发写入时可以看到什么。却忽略了两个事务并发写入的问题——我们只讨论了脏写，一种特定类型的写-写冲突是可能出现的。

并发的写入事务之间还有其他几种有趣的冲突。其中最着名的是丢失更新（**lost update**）问题，如图7-1所示，以两个并发计数器增量为例。

如果应用从数据库中读取一些值，修改它并写回修改的值（读取-修改-写入序列），则可能会发生丢失更新的问题。如果两个事务同时执行，则其中一个的修改可能会丢失，因为第二个写入的内容并没有包括第一个事务的修改（有时会说后面写入狠揍（**clobber**）了前面的写入）这种模式发生在各种不同的情况下：

- 增加计数器或更新账户余额（需要读取当前值，计算新值并写回更新后的值）
- 在复杂值中进行本地修改：例如，将元素添加到JSON文档中的一个列表（需要解析文档，进行更改并写回修改的文档）
- 两个用户同时编辑wiki页面，每个用户通过将整个页面内容发送到服务器来保存其更改，覆写数据库中当前的任何内容。

这是一个普遍的问题，所以已经开发了各种解决方案。

原子写

许多数据库提供了原子更新操作，从而消除了在应用程序代码中执行读取-修改-写入序列的需要。如果你的代码可以用这些操作来表达，那这通常是最好的解决方案。例如，下面的指令在大多数关系数据库中是并发安全的：

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

类似地，像MongoDB这样的文档数据库提供了对JSON文档的一部分进行本地修改的原子操作，Redis提供了修改数据结构（如优先级队列）的原子操作。并不是所有的写操作都可以用原子操作的方式来表达，例如维基页面的更新涉及到任意文本编辑[viii]，但是在可以使用原子操作的情况下，它们通常是最好的选择。

viii

> viii. 将文本文档的编辑表示为原子的变化流是可能的，尽管相当复杂。参阅“自动冲突解决”。↩

原子操作通常通过在读取对象时，获取其上的排它锁来实现。以便更新完成之前没有其他事务可以读取它。这种技术有时被称为游标稳定性（**cursor stability**）【36,37】。另一个选择是简单地强制所有的原子操作在单一线程上执行。

不幸的是，ORM框架很容易意外地执行不安全的读取-修改-写入序列，而不是使用数据库提供的原子操作【38】。如果你知道自己在做什么那当然不是问题，但它经常产生那种很难测出来的微妙Bug。

显式锁定

如果数据库的内置原子操作没有提供必要的功能，防止丢失更新的另一个选择是让应用程序显式地锁定将要更新的对象。然后应用程序可以执行读取-修改-写入序列，如果任何其他事务尝试同时读取同一个对象，则强制等待，直到第一个读取-修改-写入序列完成。

例如，考虑一个多人游戏，其中几个玩家可以同时移动相同的棋子。在这种情况下，一个原子操作可能是不够的，因为应用程序还需要确保玩家的移动符合游戏规则，这可能涉及到一些不能合理地用数据库查询实现的逻辑。但你可以使用锁来防止两名玩家同时移动相同的棋子，如例7-1所示。

例**7-1** 显式锁定行以防止丢失更新

```
BEGIN TRANSACTION;
SELECT * FROM figures
    WHERE name = 'robot' AND game_id = 222
FOR UPDATE;

-- 检查玩家的操作是否有效，然后更新先前SELECT返回棋子的位置。
UPDATE figures SET position = 'c4' WHERE id = 1234;
COMMIT;
```

- `FOR UPDATE` 子句告诉数据库应该对该查询返回的所有行加锁。

这是有效的，但要做对，你需要仔细考虑应用逻辑。忘记在代码某处加锁很容易引入竞争条件。

自动检测丢失的更新

原子操作和锁是通过强制读取-修改-写入序列按顺序发生，来防止丢失更新的方法。另一种方法是允许它们并行执行，如果事务管理器检测到丢失更新，则中止事务并强制它们重试其读取-修改-写入序列。

这种方法的一个优点是，数据库可以结合快照隔离高效地执行此检查。事实上，PostgreSQL的可重复读，Oracle的可串行化和SQL Server的快照隔离级别，都会自动检测到丢失更新，并中止惹麻烦的事务。但是，MySQL/InnoDB的可重复读并不会检测丢失更新【23】。一些

作者【28,30】认为，数据库必须能防止丢失更新才称得上是提供了快照隔离，所以在这个定义下，MySQL下不提供快照隔离。

丢失更新检测是一个很好的功能，因为它不需要应用代码使用任何特殊的数据库功能，你可能会忘记使用锁或原子操作，从而引入错误；但丢失更新的检测是自动发生的，因此不太容易出错。

比较并设置（CAS）

在不提供事务的数据库中，有时会发现一种原子操作：比较并设置（**CAS, Compare And Set**）（先前在“单对象写入”中提到）。此操作的目的是为了避免丢失更新：只有当前值从上次读取时一直未改变，才允许更新发生。如果当前值与先前读取的值不匹配，则更新不起作用，且必须重试读取-修改-写入序列。

例如，为了防止两个用户同时更新同一个wiki页面，可以尝试类似这样的方式，只有当用户开始编辑页面内容时，才会发生更新：

```sql
-- 根据数据库的实现情况，这可能也可能不安全
UPDATE wiki_pages SET content = '新内容'
  WHERE id = 1234 AND content = '旧内容';
```

如果内容已经更改并且不再与“旧内容”相匹配，则此更新将不起作用，因此您需要检查更新是否生效，必要时重试。但是，如果数据库允许 `WHERE` 子句从旧快照中读取，则此语句可能无法防止丢失更新，因为即使发生了另一个并发写入，`WHERE` 条件也可能为真。在依赖数据库的CAS操作前要检查其是否安全。

冲突解决和复制

在复制数据库中（参见第5章），防止丢失的更新需要考虑另一个维度：由于在多个节点上存在数据副本，并且在不同节点上的数据可能被并发地修改，因此需要采取一些额外的步骤来防止丢失更新。

锁和CAS操作假定有一个最新的数据副本。但是多主或无主复制的数据库通常允许多个写入并发执行，并异步复制到副本上，因此无法保证有一份数据的最新副本。所以基于锁或CAS操作的技术不适用于这种情况。（我们将在“线性化”中更详细地讨论这个问题。）

相反，如“检测并发写入”一节所述，这种复制数据库中的一种常见方法是允许并发写入创建多个冲突版本的值（也称为兄弟），并使用应用代码或特殊数据结构在事实发生之后解决和合并这些版本。

原子操作可以在复制的上下文中很好地工作，尤其当它们具有可交换性时（即，可以在不同的副本上以不同的顺序应用它们，且仍然可以得到相同的结果）。例如，递增计数器或向集合添加元素是可交换的操作。这是Riak 2.0数据类型背后的思想，它可以防止复制副本丢失更新。当不同的客户端同时更新一个值时，Riak自动将更新合并在一起，以免丢失更新【39】。

另一方面，最后写入为准（LWW）的冲突解决方法很容易丢失更新，如"最后写入为准（丢弃并发写入）"中所述。不幸的是，LWW是许多复制数据库中的默认值。

写入偏差与幻读

前面的章节中，我们看到了脏写和丢失更新，当不同的事务并发地尝试写入相同的对象时，会出现两种竞争条件。为了避免数据损坏，这些竞争条件需要被阻止——既可以由数据库自动执行，也可以通过锁和原子写操作这类手动安全措施来防止。

但是，并发写入间可能发生的竞争条件还没有完。在本节中，我们将看到一些更微妙的冲突例子。

首先，想象一下这个例子：你正在为医院写一个医生轮班管理程序。医院通常会同时要求几位医生待命，但底线是至少有一位医生在待命。医生可以放弃他们的班次（例如，如果他们自己生病了），只要至少有一个同事在这一班中继续工作【40,41】。

现在想象一下，Alice和Bob是两位值班医生。两人都感到不适，所以他们都决定请假。不幸的是，他们恰好在同一时间点击按钮下班。图7-8说明了接下来的事情。

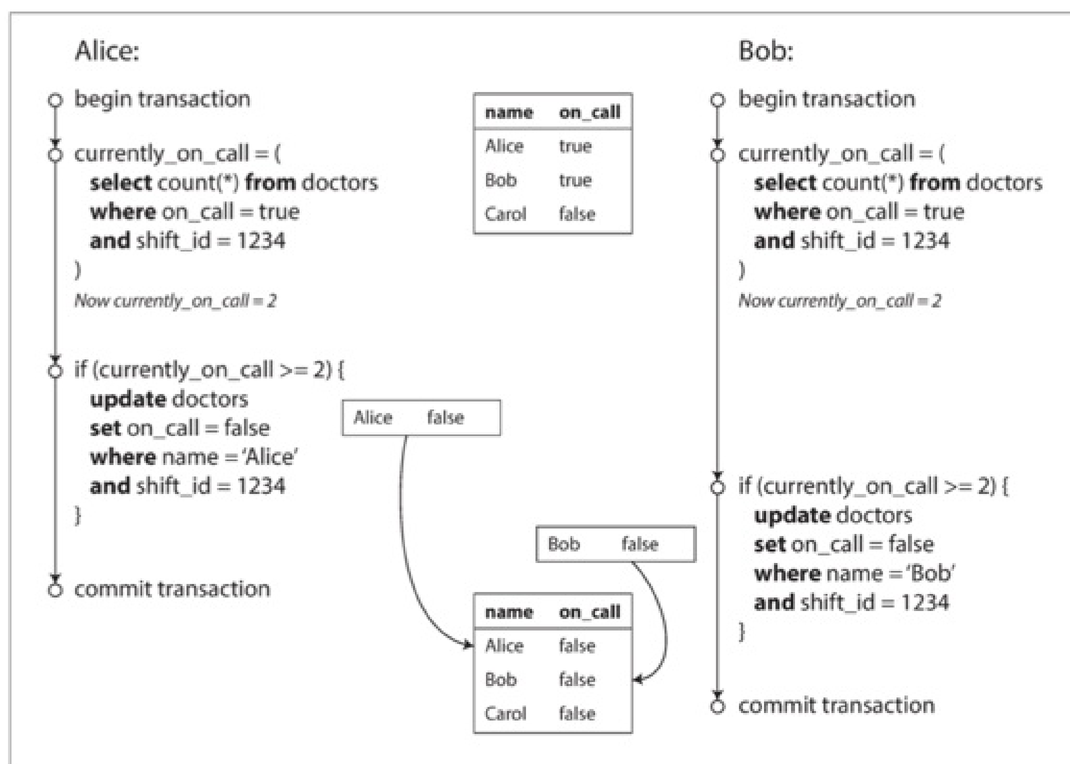


图**7-8** 写入偏差导致应用程序错误的示例

在两个事务中，应用首先检查是否有两个或以上的医生正在值班；如果是的话，它就假定一名医生可以安全地休班。由于数据库使用快照隔离，两次检查都返回 2 ，所以两个事务都进入下一个阶段。Alice更新自己的记录休班了，而Bob也做了一样的事情。两个事务都成功提交了，现在没有医生值班了。违反了至少有一名医生在值班的要求。

写偏差的特征

这种异常称为写偏差【28】。它既不是脏写，也不是丢失更新，因为这两个事务正在更新两个不同的对象（Alice和Bob各自的待命记录）。在这里发生的冲突并不是那么明显，但是这显然是一个竞争条件：如果两个事务一个接一个地运行，那么第二个医生就不能歇班了。异常行为只有在事务并发进行时才有可能。

可以将写入偏差视为丢失更新问题的一般化。如果两个事务读取相同的对象，然后更新其中一些对象（不同的事务可能更新不同的对象），则可能发生写入偏差。在多个事务更新同一个对象的特殊情况下，就会发生脏写或丢失更新（取决于时机）。

我们看到，有各种不同的方法来防止丢失的更新。随着写偏差，我们的选择更受限制：

- 由于涉及多个对象，单对象的原子操作不起作用。
- 不幸的是，在一些快照隔离的实现中，自动检测丢失更新对此并没有帮助。在PostgreSQL的可重复读，MySQL/InnoDB的可重复读，Oracle可序列化或SQL Server的快照隔离级别中，都不会自动检测写入偏差【23】。自动防止写入偏差需要真正的可序列化隔离（请参见“可序列化”）。
- 某些数据库允许配置约束，然后由数据库强制执行（例如，唯一性，外键约束或特定值限制）。但是为了指定至少有一名医生必须在线，需要一个涉及多个对象的约束。大多数数据库没有内置对这种约束的支持，但是你可以使用触发器，或者物化视图来实现它们，这取决于不同的数据库【42】。
- 如果无法使用可序列化的隔离级别，则此情况下的次优选项可能是显式锁定事务所依赖的行。在例子中，你可以写下如下的代码：

```
BEGIN TRANSACTION;
SELECT * FROM doctors
  WHERE on_call = TRUE
  AND shift_id = 1234 FOR UPDATE;

UPDATE doctors
  SET on_call = FALSE
  WHERE name = 'Alice'
  AND shift_id = 1234;

COMMIT;
```

- 和以前一样，`FOR UPDATE` 告诉数据库锁定返回的所有行用于更新。

写偏差的更多例子

写偏差乍看像是一个深奥的问题，但一旦意识到这一点，很容易会注意到更多可能的情况。以下是一些例子：

会议室预订系统

假设你想强制执行，同一时间不能同时在两个会议室预订【43】。当有人想要预订时，首先检查是否存在相互冲突的预订（即预订时间范围重叠的同一房间），如果没有找到，则创建会议（请参见示例7-2）[ix]。

ix

> ix. 在PostgreSQL中，您可以使用范围类型优雅地执行此操作，但在其他数据库中并未得到广泛支持。↩

例7-2 会议室预订系统试图避免重复预订（在快照隔离下不安全）

```
BEGIN TRANSACTION;

-- 检查所有现存的与12:00~13:00重叠的预定
SELECT COUNT(*) FROM bookings
WHERE room_id = 123 AND
    end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';

-- 如果之前的查询返回0
INSERT INTO bookings(room_id, start_time, end_time, user_id)
  VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

不幸的是，快照隔离并不能防止另一个用户同时插入冲突的会议。为了确保不会遇到调度冲突，你又需要可序列化的隔离级别了。

多人游戏

在例7-1中，我们使用一个锁来防止丢失更新（也就是确保两个玩家不能同时移动同一个棋子）。但是锁定并不妨碍玩家将两个不同的棋子移动到棋盘上的相同位置，或者采取其他违反游戏规则的行为。按照您正在执行的规则类型，也许可以使用唯一约束，否则您很容易发生写入偏差。

抢注用户名

在每个用户拥有唯一用户名的网站上，两个用户可能会尝试同时创建具有相同用户名的帐户。可以在事务检查名称是否被抢占，如果没有则使用该名称创建账户。但是像在前面的例子中那样，在快照隔离下这是不安全的。幸运的是，唯一约束是一个简单的解决办法（第二个事务在提交时会因为违反用户名唯一约束而被中止）。

防止双重开支

允许用户花钱或积分的服务，需要检查用户的支付数额不超过其余额。可以通过在用户的帐户中插入一个试探性的消费项目来实现这一点，列出帐户中的所有项目，并检查总和是否为正值【44】。有了写入偏差，可能会发生两个支出项目同时插入，一起导致余额变为负值，但这两个事务都不会注意到另一个。

导致写入偏差的幻读

所有这些例子都遵循类似的模式：

1. 一个 `SELECT` 查询找出符合条件的行，并检查是否符合一些要求。（例如：至少有两名医生在值班；不存在对该会议室同一时段的预定；棋盘上的位置没有被其他棋子占据；用户名还没有被抢注；账户里还有足够余额）

2. 按照第一个查询的结果，应用代码决定是否继续。（可能会继续操作，也可能中止并报错）

3. 如果应用决定继续操作，就执行写入（插入、更新或删除），并提交事务。

   这个写入的效果改变了步骤2 中的先决条件。换句话说，如果在提交写入后，重复执行一次步骤1 的SELECT查询，将会得到不同的结果。因为写入改变符合搜索条件的行集（现在少了一个医生值班，那时候的会议室现在已经被预订了，棋盘上的这个位置已经被占据了，用户名已经被抢注，账户余额不够了）。

这些步骤可能以不同的顺序发生。例如可以首先进行写入，然后进行SELECT查询，最后根据查询结果决定是放弃还是提交。

在医生值班的例子中，在步骤3中修改的行，是步骤1中返回的行之一，所以我们可以通过锁定步骤1 中的行（ `SELECT FOR UPDATE` ）来使事务安全并避免写入偏差。但是其他四个例子是不同的：它们检查是否不存在某些满足条件的行，写入会添加一个匹配相同条件的行。如果步骤1中的查询没有返回任何行，则 `SELECT FOR UPDATE` 锁不了任何东西。

这种效应：一个事务中的写入改变另一个事务的搜索查询的结果，被称为幻读【3】。快照隔离避免了只读查询中幻读，但是在像我们讨论的例子那样的读写事务中，幻影会导致特别棘手的写歪斜情况。

物化冲突

如果幻读的问题是没有对象可以加锁，也许可以人为地在数据库中引入一个锁对象？

例如，在会议室预订的场景中，可以想象创建一个关于时间槽和房间的表。此表中的每一行对应于特定时间段（例如15分钟）的特定房间。可以提前插入房间和时间的所有可能组合行（例如接下来的六个月）。

现在，要创建预订的事务可以锁定（ `SELECT FOR UPDATE` ）表中与所需房间和时间段对应的行。在获得锁定之后，它可以检查重叠的预订并像以前一样插入新的预订。请注意，这个表并不是用来存储预订相关的信息——它完全就是一组锁，用于防止同时修改同一房间和时间范围内的预订。

这种方法被称为物化冲突（**materializing conflicts**），因为它将幻读变为数据库中一组具体行上的锁冲突【11】。不幸的是，弄清楚如何物化冲突可能很难，也很容易出错，而让并发控制机制泄漏到应用数据模型是很丑陋的做法。出于这些原因，如果没有其他办法可以实现，物化冲突应被视为最后的手段。在大多数情况下。可序列化（**Serializable**）的隔离级别是更可取的。

可序列化

在本章中，已经看到了几个易于出现竞争条件的事务例子。读已提交和快照隔离级别会阻止某些竞争条件，但不会阻止另一些。我们遇到了一些特别棘手的例子，写入偏差和幻读。这是一个可悲的情况：

- 隔离级别难以理解，并且在不同的数据库中实现的不一致（例如，“可重复读”的含义天差地别）。
- 光检查应用代码很难判断在特定的隔离级别运行是否安全。 特别是在大型应用程序中，您可能并不知道并发发生的所有事情。
- 没有检测竞争条件的好工具。原则上来说，静态分析可能会有帮助【26】，但研究中的技术还没法实际应用。并发问题的测试是很难的，因为它们通常是非确定性的 —— 只有在倒霉的时机下才会出现问题。

这不是一个新问题，从20世纪70年代以来就一直是这样了，当时首先引入了较弱的隔离级别【2】。一直以来，研究人员的答案都很简单：使用可序列化（**serializable**）的隔离级别！

可序列化（**Serializability**）隔离通常被认为是最强的隔离级别。它保证即使事务可以并行执行，最终的结果也是一样的，就好像它们没有任何并发性，连续挨个执行一样。因此数据库保证，如果事务在单独运行时正常运行，则它们在并发运行时继续保持正确 —— 换句话说，数据库可以防止所有可能的竞争条件。

但如果可序列化隔离级别比弱隔离级别的烂摊子要好得多，那为什么没有人见人爱？为了回答这个问题，我们需要看看实现可序列化的选项，以及它们如何执行。目前大多数提供可序列化的数据库都使用了三种技术之一，本章的剩余部分将会介绍这些技术。

- 字面意义上地串行顺序执行事务（参见“真的串行执行”）
- 两相锁定（**2PL, two-phase locking**），几十年来唯一可行的选择。（参见“两相锁定（2PL）”）
- 乐观并发控制技术，例如可序列化的快照隔离（**serializable snapshot isolation**）（参阅“可序列化的快照隔离（SSI）”

现在将主要在单节点数据库的背景下讨论这些技术；在第9章中，我们将研究如何将它们推广到涉及分布式系统中多个节点的事务。

真的串行执行

避免并发问题的最简单方法就是完全不要并发：在单个线程上按顺序一次只执行一个事务。这样做就完全绕开了检测/防止事务间冲突的问题，由此产生的隔离，正是可序列化的定义。

尽管这似乎是一个明显的主意，但数据库设计人员只是在2007年左右才决定，单线程循环执行事务是可行的【45】。如果多线程并发在过去的30年中被认为是获得良好性能的关键所在，那么究竟是什么改变致使单线程执行变为可能呢？

两个进展引发了这个反思：

- RAM足够便宜了，许多场景现在都可以将完整的活跃数据集保存在内存中。（参阅“在内

存中存储一切”）。当事务需要访问的所有数据都在内存中时，事务处理的执行速度要比等待数据从磁盘加载时快得多。

- 数据库设计人员意识到OLTP事务通常很短，而且只进行少量的读写操作（参阅“事务处理或分析？”）。相比之下，长时间运行的分析查询通常是只读的，因此它们可以在串行执行循环之外的一致快照（使用快照隔离）上运行。

串行执行事务的方法在VoltDB/H-Store，Redis和Datomic中实现【46,47,48】。设计用于单线程执行的系统有时可以比支持并发的系统更好，因为它可以避免锁的协调开销。但是其吞吐量仅限于单个CPU核的吞吐量。为了充分利用单一线程，需要与传统形式不同的结构的事务。

在存储过程中封装事务

在数据库的早期阶段，意图是数据库事务可以包含整个用户活动流程。例如，预订机票是一个多阶段的过程（搜索路线，票价和可用座位，决定行程，在每段行程的航班上订座，输入乘客信息，付款）。数据库设计者认为，如果整个过程是一个事务，那么它就可以被原子化地执行。

不幸的是，人类做出决定和回应的速度非常缓慢。如果数据库事务需要等待来自用户的输入，则数据库需要支持潜在的大量并发事务，其中大部分是空闲的。大多数数据库不能高效完成这项工作，因此几乎所有的OLTP应用程序都避免在事务中等待交互式的用户输入，以此来保持事务的简短。在Web上，这意味着事务在同一个HTTP请求中被提交——一个事务不会跨越多个请求。一个新的HTTP请求开始一个新的事务。

即使人类已经找到了关键路径，事务仍然以交互式的客户端/服务器风格执行，一次一个语句。应用程序进行查询，读取结果，可能根据第一个查询的结果进行另一个查询，依此类推。查询和结果在应用程序代码（在一台机器上运行）和数据库服务器（在另一台机器上）之间来回发送。

在这种交互式的事务方式中，应用程序和数据库之间的网络通信耗费了大量的时间。如果不允许在数据库中进行并发处理，且一次只处理一个事务，则吞吐量将会非常糟糕，因为数据库大部分的时间都花费在等待应用程序发出当前事务的下一个查询。在这种数据库中，为了获得合理的性能，需要同时处理多个事务。

出于这个原因，具有单线程串行事务处理的系统不允许交互式的多语句事务。取而代之，应用程序必须提前将整个事务代码作为存储过程提交给数据库。这些方法之间的差异如图7-9 所示。如果事务所需的所有数据都在内存中，则存储过程可以非常快地执行，而不用等待任何网络或磁盘I/O。

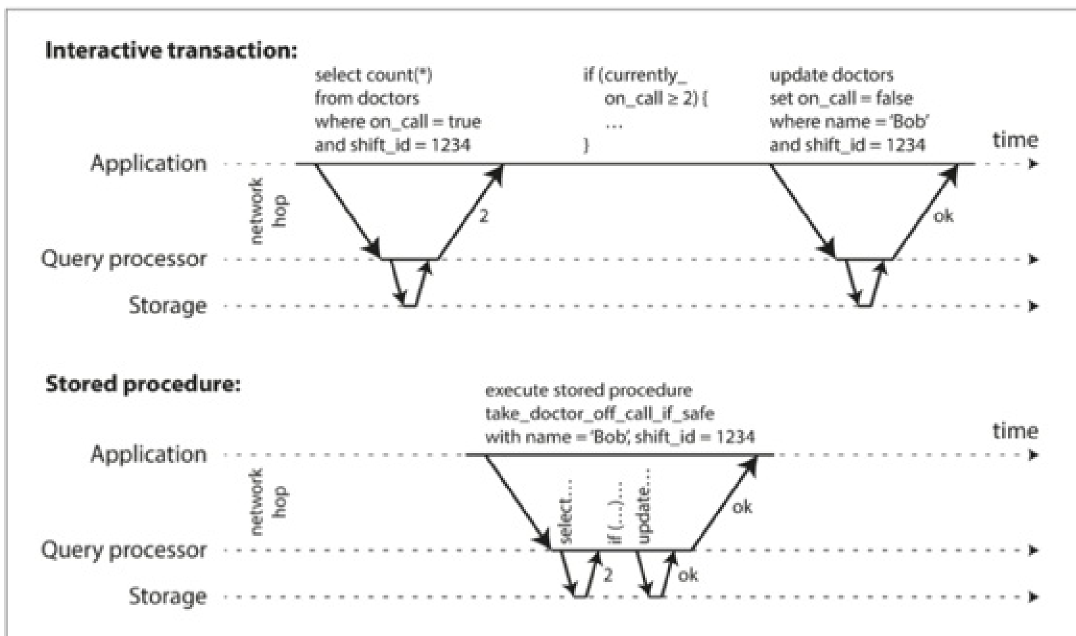


图**7-9** 交互式事务和存储过程之间的区别（使用图**7-8**的示例事务）

存储过程的优点和缺点

存储过程在关系型数据库中已经存在了一段时间了，自1999年以来它们一直是SQL标准（SQL/PSM）的一部分。出于各种原因，它们的名声有点不太好：

- 每个数据库厂商都有自己的存储过程语言（Oracle有PL/SQL，SQL Server有T-SQL，PostgreSQL有PL/pgSQL等）。这些语言并没有跟上通用编程语言的发展，所以从今天的角度来看，它们看起来相当丑陋和陈旧，而且缺乏大多数编程语言中能找到的库的生态系统。
- 与应用服务器相，比在数据库中运行的管理困难，调试困难，版本控制和部署起来也更为尴尬，更难测试，更难和用于监控的指标收集系统相集成。
- 数据库通常比应用服务器对性能敏感的多，因为单个数据库实例通常由许多应用服务器共享。数据库中一个写得不好的存储过程（例如，占用大量内存或CPU时间）会比在应用服务器中相同的代码造成更多的麻烦。

但是这些问题都是可以克服的。现代的存储过程实现放弃了PL/SQL，而是使用现有的通用编程语言：VoltDB使用Java或Groovy，Datomic使用Java或Clojure，而Redis使用Lua。

存储过程与内存存储，使得在单个线程上执行所有事务变得可行。由于不需要等待I/O，且避免了并发控制机制的开销，它们可以在单个线程上实现相当好的吞吐量。

VoltDB还使用存储过程进行复制：但不是将事务的写入结果从一个节点复制到另一个节点，而是在每个节点上执行相同的存储过程。因此VoltDB要求存储过程是确定性的（在不同的节点上运行时，它们必须产生相同的结果）。举个例子，如果事务需要使用当前的日期和时间，则必须通过特殊的确定性API来实现。

分区

顺序执行所有事务使并发控制简单多了，但数据库的事务吞吐量被限制为单机单核的速度。只读事务可以使用快照隔离在其它地方执行，但对于写入吞吐量较高的应用，单线程事务处理器可能成为一个严重的瓶颈。

为了扩展到多个CPU核心和多个节点，可以对数据进行分区（参见第6章），在VoltDB中支持这样做。如果你可以找到一种对数据集进行分区的方法，以便每个事务只需要在单个分区中读写数据，那么每个分区就可以拥有自己独立运行的事务处理线程。在这种情况下可以为每个分区指派一个独立的CPU核，事务吞吐量就可以与CPU核数保持线性扩展【47】。

但是，对于需要访问多个分区的任何事务，数据库必须在触及的所有分区之间协调事务。存储过程需要跨越所有分区锁定执行，以确保整个系统的可串行性。

由于跨分区事务具有额外的协调开销，所以它们比单分区事务慢得多。 VoltDB报告的吞吐量大约是每秒1000个跨分区写入，比单分区吞吐量低几个数量级，并且不能通过增加更多的机器来增加【49】。

事务是否可以是划分至单个分区很大程度上取决于应用数据的结构。简单的键值数据通常可以非常容易地进行分区，但是具有多个二级索引的数据可能需要大量的跨分区协调（参阅“分片与次级索引”）。

串行执行小结

在特定约束条件下，真的串行执行事务，已经成为一种实现可序列化隔离等级的可行办法。

- 每个事务都必须小而快，只要有一个缓慢的事务，就会拖慢所有事务处理。
- 仅限于活跃数据集可以放入内存的情况。很少访问的数据可能会被移动到磁盘，但如果需要在单线程执行的事务中访问，系统就会变得非常慢[x]。
- 写入吞吐量必须低到能在单个CPU核上处理，如若不然，事务需要能划分至单个分区，且不需要跨分区协调。
- 跨分区事务是可能的，但是它们的使用程度有很大的限制。

> x. 如果事务需要访问不在内存中的数据，最好的解决方案可能是中止事务，异步地将数据提取到内存中，同时继续处理其他事务，然后在数据加载完毕时重新启动事务。这种方法被称为反缓存（**anti-caching**），正如前面在第88页“将所有内容保存在内存”中所述。 ↩

两阶段锁定（2PL）

大约30年来，在数据库中只有一种广泛使用的序列化算法：两阶段锁定（**2PL，two-phase locking**）[xi]

> xi. 有时也称为严格两阶段锁定（**SS2PL, strict two-phas locking**），以便和其他2PL变体区分。 ↩

> **2PL不是2PC**
>
> 请注意，虽然两阶段锁定（2PL）听起来非常类似于两阶段提交（2PC），但它们是完全不同的东西。我们将在第9章讨论2PC。

之前我们看到锁通常用于防止脏写（参阅“没有脏写”一节）：如果两个事务同时尝试写入同一个对象，则锁可确保第二个写入必须等到第一个写入完成事务（中止或提交），然后才能继续。

两阶段锁定定类似，但使锁的要求更强。只要没有写入，就允许多个事务同时读取同一个对象。但对象只要有写入（修改或删除），就需要独占访问（**exclusive access**）权限：

- 如果事务A读取了一个对象，并且事务B想要写入该对象，那么B必须等到A提交或中止才能继续。（这确保B不能在A底下意外地改变对象。）
- 如果事务A写入了一个对象，并且事务B想要读取该对象，则B必须等到A提交或中止才能继续。（像图7-1那样读取旧版本的对象在2PL下是不可接受的。）

在2PL中，写入不仅会阻塞其他写入，也会阻塞读，反之亦然。快照隔离使得读不阻塞写，写也不阻塞读（参阅“实现快照隔离”），这是2PL和快照隔离之间的关键区别。另一方面，因为2PL提供了可序列化的性质，它可以防止早先讨论的所有竞争条件，包括丢失更新和写入偏差。

实现两阶段锁

2PL用于MySQL（InnoDB）和SQL Server中的可序列化隔离级别，以及DB2中的可重复读隔离级别【23,36】。

读与写的阻塞是通过为数据库中每个对象添加锁来实现的。锁可以处于共享模式（**shared mode**）或独占模式（**exclusive mode**）。锁使用如下：

- 若事务要读取对象，则须先以共享模式获取锁。允许多个事务同时持有共享锁。但如果另一个事务已经在对象上持有排它锁，则这些事务必须等待。
- 若事务要写入一个对象，它必须首先以独占模式获取该锁。没有其他事务可以同时持有锁（无论是共享模式还是独占模式），所以如果对象上存在任何锁，该事务必须等待。
- 如果事务先读取再写入对象，则它可能会将其共享锁升级为独占锁。升级锁的工作与直接获得排他锁相同。
- 事务获得锁之后，必须继续持有锁直到事务结束（提交或中止）。这就是“两阶段”这个名字的来源：第一阶段（当事务正在执行时）获取锁，第二阶段（在事务结束时）释放所有的锁。

由于使用了这么多的锁，因此很可能会发生：事务A等待事务B释放它的锁，反之亦然。这种情况叫做死锁（**Deadlock**）。数据库会自动检测事务之间的死锁，并中止其中一个，以便另一个继续执行。被中止的事务需要由应用程序重试。

两阶段锁定的性能

两阶段锁定的巨大缺点，以及70年代以来没有被所有人使用的原因，是其性能问题。两阶段锁定下的事务吞吐量与查询响应时间要比弱隔离级别下要差得多。

这一部分是由于获取和释放所有这些锁的开销，但更重要的是由于并发性的降低。按照设计，如果两个并发事务试图做任何可能导致竞争条件的事情，那么必须等待另一个完成。

传统的关系数据库不限制事务的持续时间，因为它们是为等待人类输入的交互式应用而设计的。因此，当一个事务需要等待另一个事务时，等待的时长并没有限制。即使你保证所有的事务都很短，如果有多个事务想要访问同一个对象，那么可能会形成一个队列，所以事务可能需要等待几个其他事务才能完成。

因此，运行2PL的数据库可能具有相当不稳定的延迟，如果在工作负载中存在争用，那么可能高百分位点处的响应会非常的慢（参阅“描述性能”）。可能只需要一个缓慢的事务，或者一个访问大量数据并获取许多锁的事务，就能把系统的其他部分拖慢，甚至迫使系统停机。当需要稳健的操作时，这种不稳定性是有问题的。

基于锁实现的读已提交隔离级别可能发生死锁，但在基于2PL实现的可序列化隔离级别中，它们会出现的频繁的多（取决于事务的访问模式）。这可能是一个额外的性能问题：当事务由于死锁而被中止并被重试时，它需要从头重做它的工作。如果死锁很频繁，这可能意味着巨大的浪费。

谓词锁

在前面关于锁的描述中，我们掩盖了一个微妙而重要的细节。在“导致写入偏差的幻读”中，我们讨论了幻读（**phantoms**）的问题。即一个事务改变另一个事务的搜索查询的结果。具有可序列化隔离级别的数据库必须防止幻读。

在会议室预订的例子中，这意味着如果一个事务在某个时间窗口内搜索了一个房间的现有预订（见例7-2），则另一个事务不能同时插入或更新同一时间窗口与同一房间的另一个预订（可以同时插入其他房间的预订，或在不影响另一个预定的条件下预定同一房间的其他时间段）。

如何实现这一点？从概念上讲，我们需要一个谓词锁（**predicate lock**）【3】。它类似于前面描述的共享/排它锁，但不属于特定的对象（例如，表中的一行），它属于所有符合某些搜索条件的对象，如：

```
SELECT * FROM bookings
WHERE room_id = 123 AND
      end_time > '2018-01-01 12:00' AND
      start_time < '2018-01-01 13:00';
```

谓词锁限制访问，如下所示：

- 如果事务A想要读取匹配某些条件的对象，就像在这个 `SELECT` 查询中那样，它必须获取查询条件上的共享谓词锁（**shared-mode predicate lock**）。如果另一个事务B持有任何满足这一查询条件对象的排它锁，那么A必须等到B释放它的锁之后才允许进行查询。

- 如果事务A想要插入，更新或删除任何对象，则必须首先检查旧值或新值是否与任何现有的谓词锁匹配。如果事务B持有匹配的谓词锁，那么A必须等到B已经提交或中止后才能继续。

这里的关键思想是，谓词锁甚至适用于数据库中尚不存在，但将来可能会添加的对象（幻象）。如果两阶段锁定包含谓词锁，则数据库将阻止所有形式的写入偏差和其他竞争条件，因此其隔离实现了可串行化。

索引范围锁

不幸的是谓词锁性能不佳：如果活跃事务持有很多锁，检查匹配的锁会非常耗时。因此，大多数使用2PL的数据库实际上实现了索引范围锁（也称为间隙锁（**next-key locking**）），这是一个简化的近似版谓词锁【41,50】。

通过使谓词匹配到一个更大的集合来简化谓词锁是安全的。例如，如果你有在中午和下午1点之间预订123号房间的谓词锁，则锁定123号房间的所有时间段，或者锁定12:00~13:00时间段的所有房间（不只是123号房间）是一个安全的近似，因为任何满足原始谓词的写入也一定会满足这种更松散的近似。

在房间预订数据库中，您可能会在`room_id`列上有一个索引，并且/或者在`start_time` 和`end_time`上有索引（否则前面的查询在大型数据库上的速度会非常慢）：

- 假设您的索引位于`room_id`上，并且数据库使用此索引查找123号房间的现有预订。现在数据库可以简单地将共享锁附加到这个索引项上，指示事务已搜索123号房间用于预订。
- 或者，如果数据库使用基于时间的索引来查找现有预订，那么它可以将共享锁附加到该索引中的一系列值，指示事务已经将12:00~13:00时间段标记为用于预定。

无论哪种方式，搜索条件的近似值都附加到其中一个索引上。现在，如果另一个事务想要插入，更新或删除同一个房间和/或重叠时间段的预订，则它将不得不更新索引的相同部分。在这样做的过程中，它会遇到共享锁，它将被迫等到锁被释放。

这种方法能够有效防止幻读和写入偏差。索引范围锁并不像谓词锁那样精确（它们可能会锁定更大范围的对象，而不是维持可串行化所必需的范围），但是由于它们的开销较低，所以是一个很好的折衷。

如果没有可以挂载间隙锁的索引，数据库可以退化到使用整个表上的共享锁。这对性能不利，因为它会阻止所有其他事务写入表格，但这是一个安全的回退位置。

序列化快照隔离（SSI）

本章描绘了数据库中并发控制的黯淡画面。一方面，我们实现了性能不好（2PL）或者扩展性不好（串行执行）的可序列化隔离级别。另一方面，我们有性能良好的弱隔离级别，但容易出现各种竞争条件（丢失更新，写入偏差，幻读等）。序列化的隔离级别和高性能是从根本上相互矛盾的吗？

也许不是：一个称为可序列化快照隔离（**SSI, serializable snapshot isolation**）的算法是非常有前途的。它提供了完整的可序列化隔离级别，但与快照隔离相比只有只有很小的性能损失。 SSI是相当新的：它在2008年首次被描述【40】，并且是Michael Cahill的博士论文【51】的主题。

今天，SSI既用于单节点数据库（PostgreSQL9.1 以后的可序列化隔离级别）和分布式数据库（FoundationDB使用类似的算法）。由于SSI与其他并发控制机制相比还很年轻，还处于在实践中证明自己表现的阶段。但它有可能因为足够快而在未来成为新的默认选项。

悲观与乐观的并发控制

两阶段锁是一种所谓的悲观并发控制机制（**pessimistic**）：它是基于这样的原则：如果有事情可能出错（如另一个事务所持有的锁所表示的），最好等到情况安全后再做任何事情。这就像互斥，用于保护多线程编程中的数据结构。

从某种意义上说，串行执行可以称为悲观到了极致：在事务持续期间，每个事务对整个数据库（或数据库的一个分区）具有排它锁，作为对悲观的补偿，我们让每笔事务执行得非常快，所以只需要短时间持有“锁”。

相比之下，序列化快照隔离是一种乐观（**optimistic**）的并发控制技术。在这种情况下，乐观意味着，如果存在潜在的危险也不阻止事务，而是继续执行事务，希望一切都会好起来。当一个事务想要提交时，数据库检查是否有什么不好的事情发生（即隔离是否被违反）；如果是的话，事务将被中止，并且必须重试。只有可序列化的事务才被允许提交。

乐观并发控制是一个古老的想法【52】，其优点和缺点已经争论了很长时间【53】。如果存在很多争用（**contention**）（很多事务试图访问相同的对象），则表现不佳，因为这会导致很大一部分事务需要中止。如果系统已经接近最大吞吐量，来自重试事务的额外负载可能会使性能变差。

但是，如果有足够的备用容量，并且事务之间的争用不是太高，乐观的并发控制技术往往比悲观的要好。可交换的原子操作可以减少争用：例如，如果多个事务同时要增加一个计数器，那么应用增量的顺序（只要计数器不在同一个事务中读取）就无关紧要了，所以并发增量可以全部应用且无需冲突。

顾名思义，SSI基于快照隔离——也就是说，事务中的所有读取都是来自数据库的一致性快照（参见“快照隔离和可重复读取”）。与早期的乐观并发控制技术相比这是主要的区别。在快照隔离的基础上，SSI添加了一种算法来检测写入之间的序列化冲突，并确定要中止哪些事务。

基于过时前提的决策

先前讨论了快照隔离中的写入偏差（参阅“写入偏差和幻像”）时，我们观察到一个循环模式：事务从数据库读取一些数据，检查查询的结果，并根据它看到的结果决定采取一些操作（写入数据库）。但是，在快照隔离的情况下，原始查询的结果在事务提交时可能不再是最新的，因为数据可能在同一时间被修改。

换句话说，事务基于一个前提（**premise**）采取行动（事务开始时候的事实，例如：“目前有两名医生正在值班”）。之后当事务要提交时，原始数据可能已经改变——前提可能不再成立。

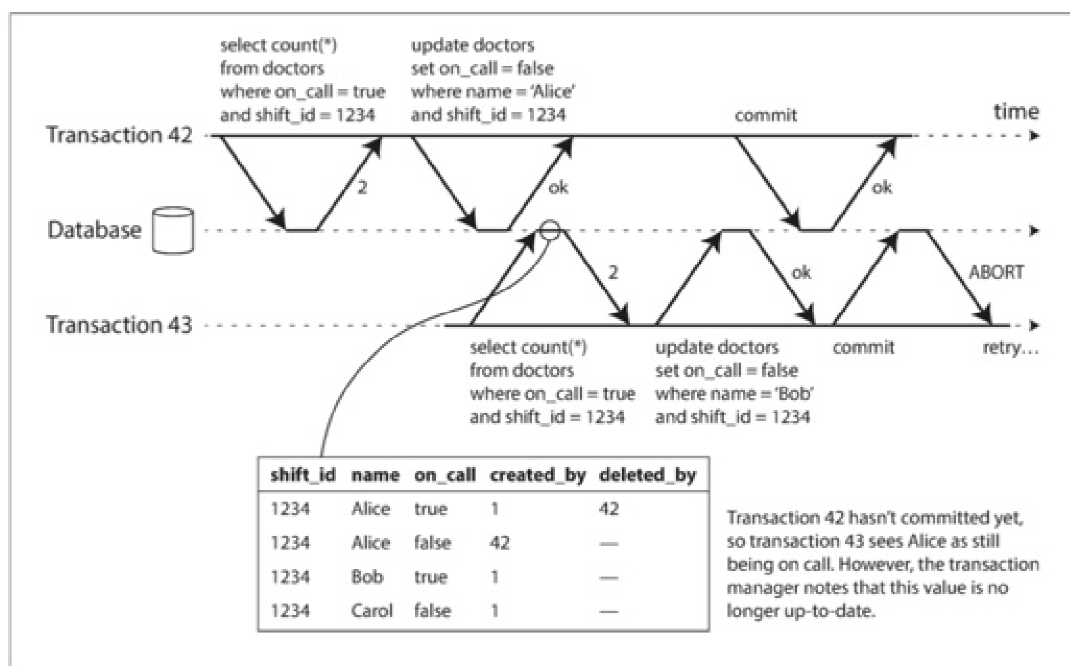
当应用程序进行查询时（例如，“当前有多少医生正在值班？”），数据库不知道应用逻辑如何使用该查询结果。在这种情况下为了安全，数据库需要假设任何对该结果集的变更都可能会使该事务中的写入变得无效。 换而言之，事务中的查询与写入可能存在因果依赖。为了提供可序列化的隔离级别，如果事务在过时的前提下执行操作，数据库必须能检测到这种情况，并中止事务。

数据库如何知道查询结果是否可能已经改变？有两种情况需要考虑：

- 检测对旧MVCC对象版本的读取（读之前存在未提交的写入）
- 检测影响先前读取的写入（读之后发生写入）

检测旧**MVCC**读取

回想一下，快照隔离通常是通过多版本并发控制（MVCC；见图7-10）来实现的。当一个事务从MVCC数据库中的一致快照读时，它将忽略取快照时尚未提交的任何其他事务所做的写入。在图7-10中，事务43 认为Alice的 `on_call = true` ，因为事务42（修改Alice的待命状态）未被提交。然而，在事务43想要提交时，事务42 已经提交。这意味着在读一致性快照时被忽略的写入已经生效，事务43 的前提不再为真。



shift_id	name	on_call	created_by	deleted_by
1234	Alice	true	1	42
1234	Alice	false	42	—
1234	Bob	true	1	—
1234	Carol	false	1	—

图**7-10** 检测事务何时从**MVCC**快照读取过时的值

为了防止这种异常，数据库需要跟踪一个事务由于MVCC可见性规则而忽略另一个事务的写入。当事务想要提交时，数据库检查是否有任何被忽略的写入现在已经被提交。如果是这样，事务必须中止。

为什么要等到提交？当检测到陈旧的读取时，为什么不立即中止事务43 ？因为如果事务43 是只读事务，则不需要中止，因为没有写入偏差的风险。当事务43 进行读取时，数据库还不知道事务是否要稍后执行写操作。此外，事务42 可能在事务43 被提交的时候中止或者可能仍然未被提交，因此读取可能终究不是陈旧的。通过避免不必要的中止，SSI 保留快照隔离对从一致快照中长时间运行的读取的支持。

检测影响之前读取的写入

第二种情况要考虑的是另一个事务在读取数据之后修改数据。这种情况如图7-11所示。

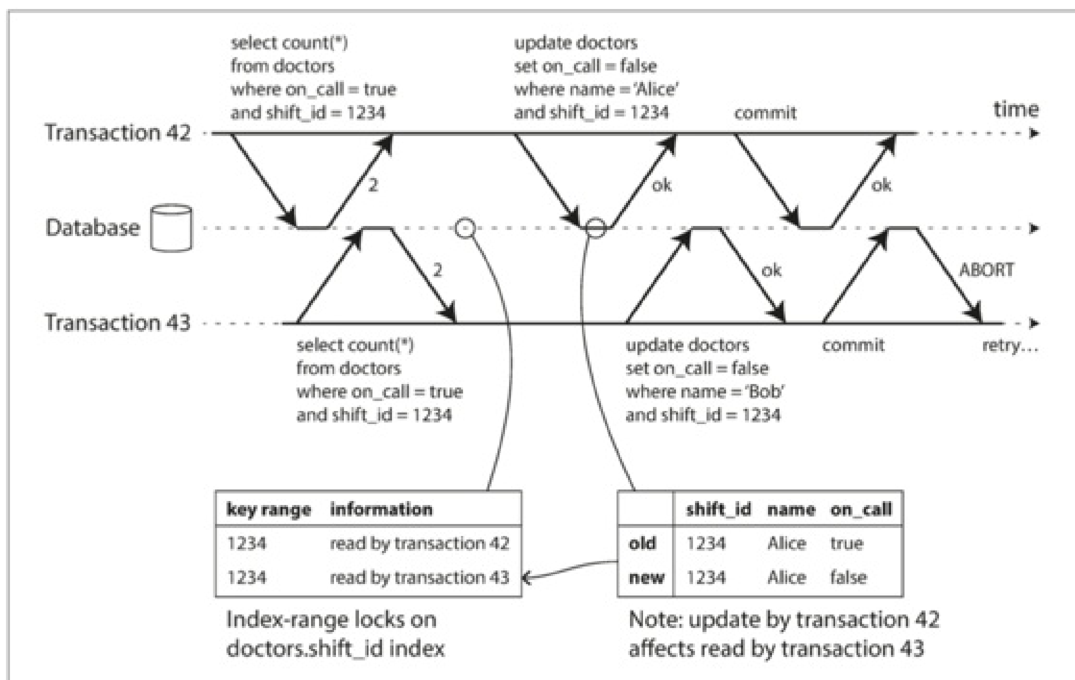


图**7-11** 在可序列化快照隔离中，检测一个事务何时修改另一个事务的读取。

在两阶段锁定的上下文中，我们讨论了索引范围锁（请参阅“索引范围锁”），它允许数据库锁定与某个搜索查询匹配的所有行的访问权，例如 `WHERE shift_id = 1234` 。可以在这里使用类似的技术，除了SSI锁不会阻塞其他事务。

在图7-11中，事务42 和43 都在班次1234 查找值班医生。如果在 `shift_id` 上有索引，则数据库可以使用索引项1234 来记录事务42 和43 读取这个数据的事实。（如果没有索引，这个信息可以在表级别进行跟踪）。这个信息只需要保留一段时间：在一个事务完成（提交或中止）之后，所有的并发事务完成之后，数据库就可以忘记它读取的数据了。

当事务写入数据库时，它必须在索引中查找最近曾读取受影响数据的其他事务。这个过程类似于在受影响的键范围上获取写锁，但锁并不会阻塞事务到其他事务完成，而是像一个引线一样只是简单通知其他事务：你们读过的数据可能不是最新的啦。

在图7-11中，事务43 通知事务42 其先前读已过时，反之亦然。事务42首先提交并成功，尽管事务43 的写影响了42 ，但因为事务43 尚未提交，所以写入尚未生效。然而当事务43 想要提交时，来自事务42 的冲突写入已经被提交，所以事务43 必须中止。

可序列化的快照隔离的性能

与往常一样，许多工程细节会影响算法的实际表现。例如一个权衡是跟踪事务的读取和写入的粒度（**granularity**）。如果数据库详细地跟踪每个事务的活动（细粒度），那么可以准确地确定哪些事务需要中止，但是簿记开销可能变得很显著。简略的跟踪速度更快（粗粒度），但可能会导致更多不必要的事务中止。

在某些情况下，事务可以读取被另一个事务覆盖的信息：这取决于发生了什么，有时可以证明执行结果无论如何都是可序列化的。 PostgreSQL使用这个理论来减少不必要的中止次数【11,41】。

与两阶段锁定相比，可序列化快照隔离的最大优点是一个事务不需要阻塞等待另一个事务所持有的锁。就像在快照隔离下一样，写不会阻塞读，反之亦然。这种设计原则使得查询延迟更可预测，变量更少。特别是，只读查询可以运行在一致的快照上，而不需要任何锁定，这对于读取繁重的工作负载非常有吸引力。

与串行执行相比，可序列化快照隔离并不局限于单个CPU核的吞吐量：FoundationDB将检测到的序列化冲突分布在多台机器上，允许扩展到很高的吞吐量。即使数据可能跨多台机器进行分区，事务也可以在保证可序列化隔离等级的同时读写多个分区中的数据【54】。

中止率显着影响SSI的整体表现。例如，长时间读取和写入数据的事务很可能会发生冲突并中止，因此SSI要求同时读写的事务尽量短（只读长事务可能没问题）。对于慢事务，SSI可能比两阶段锁定或串行执行更不敏感。

本章小结

事务是一个抽象层，允许应用程序假装某些并发问题和某些类型的硬件和软件故障不存在。各式各样的错误被简化为一种简单情况：事务中止（**transaction abort**），而应用需要的仅仅是重试。

在本章中介绍了很多问题，事务有助于防止这些问题发生。并非所有应用都易受此类问题影响：具有非常简单访问模式的应用（例如每次读写单条记录）可能无需事务管理。但是对于更复杂的访问模式，事务可以大大减少需要考虑的潜在错误情景数量。

如果没有事务处理，各种错误情况（进程崩溃，网络中断，停电，磁盘已满，意外并发等）意味着数据可能以各种方式变得不一致。例如，非规范化的数据可能很容易与源数据不同步。如果没有事务处理，就很难推断复杂的交互访问可能对数据库造成的影响。

本章深入讨论了并发控制的话题。我们讨论了几个广泛使用的隔离级别，特别是读已提交，快照隔离（有时称为可重复读）和可序列化。并通过研究竞争条件的各种例子，来描述这些隔离等级：

脏读

一个客户端读取到另一个客户端尚未提交的写入。读已提交或更强的隔离级别可以防止脏读。

脏写

一个客户端覆盖写入了另一个客户端尚未提交的写入。几乎所有的事务实现都可以防止脏写。

读取偏差（不可重复读）

在同一个事务中，客户端在不同的时间点会看见数据库的不同状态。快照隔离经常用于解决这个问题，它允许事务从一个特定时间点的一致性快照中读取数据。快照隔离通常使用多版本并发控制（**MVCC**）来实现。

更新丢失

两个客户端同时执行读取**-**修改**-**写入序列。其中一个写操作，在没有合并另一个写入变更情况下，直接覆盖了另一个写操作的结果。所以导致数据丢失。快照隔离的一些实现可以自动防止这种异常，而另一些实现则需要手动锁定（`SELECT FOR UPDATE`）。

写偏差

一个事务读取一些东西，根据它所看到的值作出决定，并将决定写入数据库。但是，写作的时候，决定的前提不再是真实的。只有可序列化的隔离才能防止这种异常。

幻读

事务读取符合某些搜索条件的对象。另一个客户端进行写入，影响搜索结果。快照隔离可以防止直接的幻像读取，但是写入歪斜环境中的幻影需要特殊处理，例如索引范围锁定。

弱隔离级别可以防止这些异常情况，但是让应用程序开发人员手动处理其他应用程序（例如，使用显式锁定）。只有可序列化的隔离才能防范所有这些问题。我们讨论了实现可序列化事务的三种不同方法：

字面意义上的串行执行

如果每个事务的执行速度非常快，并且事务吞吐量足够低，足以在单个CPU核上处理，这是一个简单而有效的选择。

两阶段锁定

数十年来，两阶段锁定一直是实现可序列化的标准方式，但是许多应用出于性能问题的考虑避免使用它。

可串行化快照隔离（***SSI***）

一个相当新的算法，避免了先前方法的大部分缺点。它使用乐观的方法，允许事务执行而无需阻塞。当一个事务想要提交时，它会进行检查，如果执行不可序列化，事务就会被中止。

本章中的示例主要是在关系数据模型的上下文中。使用关系数据模型。但是，正如在讨论中，无论使用哪种数据模型，如“多对象事务的需求”中所讨论的，事务都是重要的数据库功能。

本章主要是在单机数据库的上下文中，探讨了各种概念与想法。分布式数据库中的事务，则引入了一系列新的困难挑战，将在接下来的两章中讨论。

参考文献

1. Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al.: “A History and Evaluation of System R,” *Communications of the ACM*, volume 24, number 10, pages 632–646, October 1981. doi:10.1145/358769.358784
2. Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger: “Granularity of Locks and Degrees of Consistency in a Shared Data Base,” in *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, edited by G. M. Nijssen, pages 364–394, Elsevier/North Holland Publishing, 1976. Also in *Readings in Database Systems*, 4th edition, edited by Joseph M. Hellerstein and Michael Stonebraker, MIT Press, 2005. ISBN: 978-0-262-69314-1
3. Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger: “The Notions of Consistency and Predicate Locks in a Database System,” *Communications of the ACM*, volume 19, number 11, pages 624–633, November 1976.
4. “ACID Transactions Are Incredibly Helpful,” FoundationDB, LLC, 2013.
5. John D. Cook: “ACID Versus BASE for Database Transactions,” *johndcook.com*, July 6, 2009.
6. Gavin Clarke: “NoSQL's CAP Theorem Busters: We Don't Drop ACID,” *theregister.co.uk*, November 22, 2012.
7. Theo Härder and Andreas Reuter: “Principles of Transaction-Oriented Database Recovery,” *ACM Computing Surveys*, volume 15, number 4, pages 287–317, December 1983. doi:10.1145/289.291
8. Peter Bailis, Alan Fekete, Ali Ghodsi, et al.: “HAT, not CAP: Towards Highly Available Transactions,” at *14th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2013.
9. Armando Fox, Steven D. Gribble, Yatin Chawathe, et al.: “Cluster-Based Scalable Network Services,” at *16th ACM Symposium on Operating Systems Principles* (SOSP), October 1997.
10. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at *research.microsoft.com*.
11. Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, et al.: “Making Snapshot Isolation

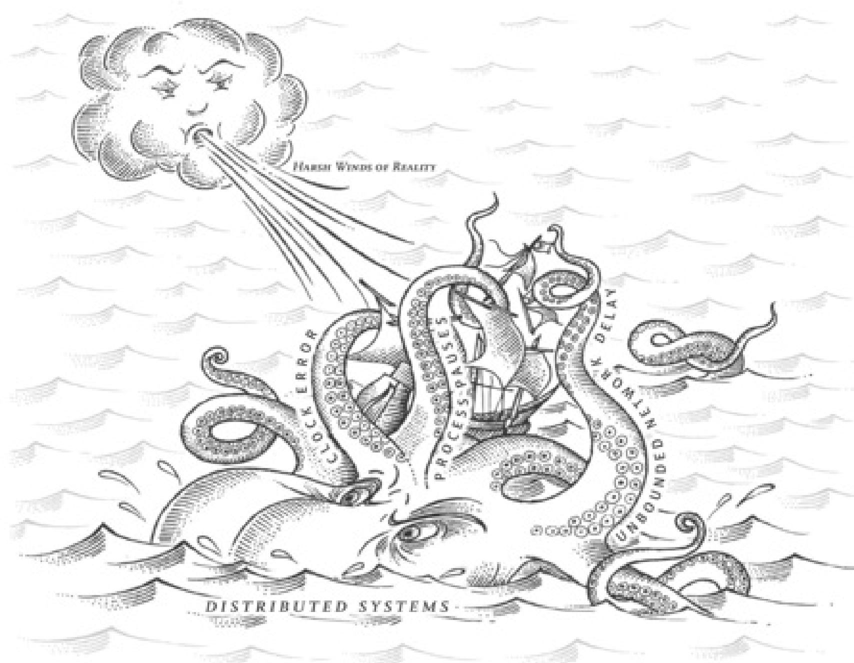
Serializable,” *ACM Transactions on Database Systems*, volume 30, number 2, pages 492–528, June 2005. doi:10.1145/1071610.1071615

12. Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge: “Understanding the Robustness of SSDs Under Power Fault,” at *11th USENIX Conference on File and Storage Technologies* (FAST), February 2013.
13. Laurie Denness: “SSDs: A Gift and a Curse,” *laur.ie*, June 2, 2015.
14. Adam Surak: “When Solid State Drives Are Not That Solid,” *blog.algolia.com*, June 15, 2015.
15. Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications,” at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.
16. Chris Siebenmann: “Unix's File Durability Problem,” *utcc.utoronto.ca*, April 14, 2016.
17. Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, et al.: “An Analysis of Data Corruption in the Storage Stack,” at *6th USENIX Conference on File and Storage Technologies* (FAST), February 2008.
18. Bianca Schroeder, Raghav Lagisetty, and Arif Merchant: “Flash Reliability in Production: The Expected and the Unexpected,” at *14th USENIX Conference on File and Storage Technologies* (FAST), February 2016.
19. Don Allison: “SSD Storage – Ignorance of Technology Is No Excuse,” *blog.korelogic.com*, March 24, 2015.
20. Dave Scherer: “Those Are Not Transactions (Cassandra 2.0),” *blog.foundationdb.com*, September 6, 2013.
21. Kyle Kingsbury: “Call Me Maybe: Cassandra,” *aphyr.com*, September 24, 2013.
22. “ACID Support in Aerospike,” Aerospike, Inc., June 2014.
23. Martin Kleppmann: “Hermitage: Testing the 'I' in ACID,” *martin.kleppmann.com*, November 25, 2014.
24. Tristan D'Agosta: “BTC Stolen from Poloniex,” *bitcointalk.org*, March 4, 2014.
25. bitcointhief2: “How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More!,” *reddit.com*, February 2, 2014.
26. Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan: “Automating the Detection of Snapshot Isolation Anomalies,” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.
27. Michael Melanson: “Transactions: The Limits of Isolation,” *michaelmelanson.net*, March 20, 2014.
28. Hal Berenson, Philip A. Bernstein, Jim N. Gray, et al.: “A Critique of ANSI SQL Isolation Levels,” at *ACM International Conference on Management of Data* (SIGMOD), May 1995.
29. Atul Adya: “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions,” PhD Thesis, Massachusetts Institute of Technology, March

1999.
30. Peter Bailis, Aaron Davidson, Alan Fekete, et al.: “Highly Available Transactions: Virtues and Limitations (Extended Version),” at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.
31. Bruce Momjian: “MVCC Unmasked,” *momjian.us*, July 2014.
32. Annamalai Gurusami: “Repeatable Read Isolation Level in InnoDB – How Consistent Read View Works,” *blogs.oracle.com*, January 15, 2013.
33. Nikita Prokopov: “Unofficial Guide to Datomic Internals,” *tonsky.me*, May 6, 2014.
34. Baron Schwartz: “Immutability, MVCC, and Garbage Collection,” *xaprb.com*, December 28, 2013.
35. J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6 Rikdeb Mukherjee: “Isolation in DB2 (Repeatable Read, Read Stability, Cursor Stability, Uncommitted Read) with Examples,” *mframes.blogspot.co.uk*, July 4, 2013.
36. Steve Hilker: “Cursor Stability (CS) – IBM DB2 Community,” *toadworld.com*, March 14, 2013.
37. Nate Wiger: “An Atomic Rant,” *nateware.com*, February 18, 2010.
38. Joel Jacobson: “Riak 2.0: Data Types,” *blog.joeljacobson.com*, March 23, 2014.
39. Michael J. Cahill, Uwe Röhm, and Alan Fekete: “Serializable Isolation for Snapshot Databases,” at *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376690
40. Dan R. K. Ports and Kevin Grittner: “Serializable Snapshot Isolation in PostgreSQL,” at *38th International Conference on Very Large Databases* (VLDB), August 2012.
41. Tony Andrews: “Enforcing Complex Constraints in Oracle,” *tonyandrews.blogspot.co.uk*, October 15, 2004.
42. Douglas B. Terry, Marvin M. Theimer, Karin Petersen, et al.: “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System,” at *15th ACM Symposium on Operating Systems Principles* (SOSP), December 1995. doi:10.1145/224056.224070
43. Gary Fredericks: “Postgres Serializability Bug,” *github.com*, September 2015.
44. Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: “The End of an Architectural Era (It’s Time for a Complete Rewrite),” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.
45. John Hugg: “H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures,” at *Data @Scale Boston*, November 2014.
46. Robert Kallman, Hideaki Kimura, Jonathan Natkins, et al.: “H-Store: A High-Performance, Distributed Main Memory Transaction Processing System,” *Proceedings of the VLDB Endowment*, volume 1, number 2, pages 1496–1499, August 2008.
47. Rich Hickey: “The Architecture of Datomic,” *infoq.com*, November 2, 2012.
48. John Hugg: “Debunking Myths About the VoltDB In-Memory Database,” *voltdb.com*, May 12, 2014.

49. Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: “Architecture of a Database System,” *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007. doi:10.1561/1900000002
50. Michael J. Cahill: “Serializable Isolation for Snapshot Databases,” PhD Thesis, University of Sydney, July 2009.
51. D. Z. Badal: “Correctness of Concurrency Control and Implications in Distributed Databases,” at *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), November 1979.
52. Rakesh Agrawal, Michael J. Carey, and Miron Livny: “Concurrency Control Performance Modeling: Alternatives and Implications,” *ACM Transactions on Database Systems* (TODS), volume 12, number 4, pages 609–654, December 1987. doi:10.1145/32204.32220
53. Dave Rosenthal: “Databases at 14.4MHz,” *blog.foundationdb.com*, December 10, 2014.

第八章：分布式系统的麻烦



> 邂逅相遇
>
> 网络延迟
>
> 存之为吾
>
> 无食我数
>
> —— Kyle Kingsbury, Carly Rae Jepsen 《网络分区的危害》（2013年）

[TOC]

最近几章中反复出现的主题是，系统如何处理错误的事情。例如，我们讨论了副本故障转移（“处理节点中断”），复制延迟（“复制延迟问题”）和事务控制（“弱隔离级别”）。当我们了解可能在实际系统中出现的各种边缘情况时，我们会更好地处理它们。

但是，尽管我们已经谈了很多错误，但之前几章仍然过于乐观。现实更加黑暗。我们现在将悲观主义最大化，假设任何可能出错的东西都会出错[i]。（经验丰富的系统运维会告诉你，这是一个合理的假设。如果你问得好，他们可能会一边治疗心理创伤一边告诉你一些可怕的故事）

> i. 除了一个例外：我们将假定故障是非拜占庭式的（参见“拜占庭故障”）。↩

使用分布式系统与在一台计算机上编写软件有着根本的区别，主要的区别在于，有许多新的和令人兴奋的方法可以使事情出错【1,2】。在这一章中，我们将了解实践中出现的问题，理解我们能够依赖，和不可以依赖的东西。

最后，作为工程师，我们的任务是构建能够完成工作的系统（即满足用户期望的保证），尽管一切都出错了。 在第9章中，我们将看看一些可以在分布式系统中提供这种保证的算法的例子。 但首先，在本章中，我们必须了解我们面临的挑战。

本章对分布式系统中可能出现的问题进行彻底的悲观和沮丧的总结。 我们将研究网络的问题（“无法访问的网络”）；时钟和时序问题（“不可靠时钟”）；我们将讨论他们可以避免的程度。所有这些问题的后果都是困惑的，所以我们将探索如何思考一个分布式系统的状态，以及如何推理发生的事情（“知识，真相和谎言”）。

故障与部分失效

当你在一台计算机上编写一个程序时，它通常会以一种相当可预测的方式运行：无论是工作还是不工作。充满错误的软件可能会让人觉得电脑有时候是“糟糕的一天”（这个问题通常是重新启动的问题），但这主要是软件写得不好的结果。

单个计算机上的软件没有根本性的不可靠原因：当硬件正常工作时，相同的操作总是产生相同的结果（这是确定性的）。如果存在硬件问题（例如，内存损坏或连接器松动），其后果通常是整个系统故障（例如，内核恐慌，“蓝屏死机”，启动失败）。装有良好软件的个人计算机通常要么功能完好，要么完全失效，而不是介于两者之间。

这是计算机设计中的一个慎重的选择：如果发生内部错误，我们宁愿电脑完全崩溃，而不是返回错误的结果，因为错误的结果很难处理。因为计算机隐藏了模糊不清的物理实现，并呈现出一个理想化的系统模型，并以数学一样的完美的方式运作。 CPU指令总是做同样的事情；如果您将一些数据写入内存或磁盘，那么这些数据将保持不变，并且不会被随机破坏。从第一台数字计算机开始，始终正确地计算这个设计目标贯穿始终【3】。

当你编写运行在多台计算机上的软件时，情况有本质上的区别。在分布式系统中，我们不再处于理想化的系统模型中，我们别无选择，只能面对现实世界的混乱现实。而在现实世界中，各种各样的事情都可能会出现问题【4】，如下面的轶事所述：

> 在我有限的经验中，我已经和很多东西打过交道：单个数据中心（DC）中长期存在的网络分区，配电单元PDU故障，开关故障，整个机架意外的电源短路，全直流主干故障，全直流电源故障，以及一个低血糖的司机把他的福特皮卡撞碎在数据中心的HVAC（加热，通风和空气）系统上。而且我甚至不是一个运维。
>
> ——柯达黑尔

在分布式系统中，尽管系统的其他部分工作正常，但系统的某些部分可能会以某种不可预知的方式被破坏。这被称为部分失效（**partial failure**）。难点在于部分失效是不确定性的（**nonderterministic**）：如果你试图做任何涉及多个节点和网络的事情，它有时可能会工

作，有时会出现不可预知的失败。正如我们将要看到的，你甚至不知道是否成功了，因为消息通过网络传播的时间也是不确定的！

这种不确定性和部分失效的可能性，使得分布式系统难以工作【5】。

云计算与超级计算机

关于如何构建大型计算系统有一系列的哲学：

- 规模的一端是高性能计算（HPC）领域。具有数千个CPU的超级计算机通常用于计算密集型科学计算任务，如天气预报或分子动力学（模拟原子和分子的运动）。
- 另一个极端是云计算（**cloud computing**），云计算并不是一个良好定义的概念【6】，但通常与多租户数据中心，连接IP网络的商品计算机（通常是以太网），弹性/按需资源分配以及计量计费等相关联。
- 传统企业数据中心位于这两个极端之间。

不同的哲学会导致不同的故障处理方式。在超级计算机中，作业通常会不时地会将计算的状态存盘到持久存储中。如果一个节点出现故障，通常的解决方案是简单地停止整个集群的工作负载。故障节点修复后，计算从上一个检查点重新开始【7,8】。因此，超级计算机更像是一个单节点计算机而不是分布式系统：通过让部分失败升级为完全失败来处理部分失败——如果系统的任何部分发生故障，只是让所有的东西都崩溃（就像单台机器上的内核恐慌一样）。

在本书中，我们将重点放在实现互联网服务的系统上，这些系统通常与超级计算机看起来有很大不同

- 许多与互联网有关的应用程序都是在线（**online**）的，因为它们需要能够随时以低延迟服务用户。使服务不可用（例如，停止群集以进行修复）是不可接受的。相比之下，像天气模拟这样的离线（批处理）工作可以停止并重新启动，影响相当小。

- 超级计算机通常由专用硬件构建而成，每个节点相当可靠，节点通过共享内存和远程直接内存访问（**RDMA**）进行通信。另一方面，云服务中的节点是由商品机器构建而成的，由于规模经济，可以以较低的成本提供相同的性能，而且具有较高的故障率。

- 大型数据中心网络通常基于IP和以太网，以闭合拓扑排列，以提供更高的二等分带宽【9】。超级计算机通常使用专门的网络拓扑结构，例如多维网格和环面 【10】，这为具有已知通信模式的HPC工作负载提供了更好的性能。

  （ 系统越大，其组件之一就越有可能发生变化。随着时间的推移，破碎的东西得到修复，新的东西被破坏，但是在一个有成千上万个节点的系统中，有理由认为总是有一些东西被破坏【7】。当错误处理策略由简单的放弃组成时，一个大的系统最终会花费大量时间从错误中恢复，而不是做有用的工作【8】。

* 如果系统可以容忍发生故障的节点，并继续保持整体工作状态，那么这对于操作和维护非常有用：例如，可以执行滚动升级（参阅第4章），一次重新启动一个节点，而服务继续服务用户不中断。在云环境中，如果一台虚拟机运行不佳，可以杀死它并请求一台新的虚拟机（希望新的虚拟机速度更快）。
* 在地理位置分散的部署中（保持数据在地理位置上接近用户以减少访问延迟），通信很可能通过互联网进行，与本地网络相比，通信速度缓慢且不可靠。超级计算机通常假设它们的所有节点都靠近在一起。

如果要使分布式系统工作，就必须接受部分故障的可能性，并在软件中建立容错机制。换句话说，我们需要从不可靠的组件构建一个可靠的系统。（正如"可靠性"中所讨论的那样，没有完美的可靠性，所以我们需要理解我们可以实际承诺的限制。）

即使在只有少数节点的小型系统中，考虑部分故障也是很重要的。在一个小系统中，很可能大部分组件在大部分时间都正常工作。然而，迟早会有一部分系统出现故障，软件必须以某种方式处理。故障处理必须是软件设计的一部分，并且作为软件的运维，您需要知道在发生故障的情况下，软件可能会表现出怎样的行为。

简单地假设缺陷很罕见，只是希望始终保持最好的状况是不明智的。考虑一系列可能的错误（甚至是不太可能的错误），并在测试环境中人为地创建这些情况来查看会发生什么是非常重要的。在分布式系统中，怀疑，悲观和偏执狂是值得的。

> ### 从不可靠的组件构建可靠的系统
>
> 您可能想知道这是否有意义——直观地看来，系统只能像其最不可靠的组件（最薄弱的环节）一样可靠。事实并非如此：事实上，从不太可靠的潜在基础构建更可靠的系统是计算机领域的一个古老思想【11】。例如：
>
> * 纠错码允许数字数据在通信信道上准确传输，偶尔会出现一些错误，例如由于无线网络上的无线电干扰【12】。
> * 互联网协议（**Internet Protocol, IP**）不可靠：可能丢弃，延迟，复制或重排数据包。传输控制协议（Transmission Control Protocol, TCP）在互联网协议（IP）之上提供了更可靠的传输层：它确保丢失的数据包被重新传输，消除重复，并且数据包被重新组装成它们被发送的顺序。
>
> 虽然这个系统可以比它的底层部分更可靠，但它的可靠性总是有限的。例如，纠错码可以处理少量的单比特错误，但是如果你的信号被干扰所淹没，那么通过信道可以得到多少数据，是有根本性的限制的【13】。TCP可以隐藏数据包的丢失，重复和重新排序，但是它不能神奇地消除网络中的延迟。
>
> 虽然更可靠的高级系统并不完美，但它仍然有用，因为它处理了一些棘手的低级错误，所以其余的错误通常更容易推理和处理。我们将在"数据库端到端的争论"中进一步探讨这个问题。

不可靠的网络

正如在第二部分的介绍中所讨论的那样，我们在本书中关注的分布式系统是无共享的系统，即通过网络连接的一堆机器。网络是这些机器可以通信的唯一途径——我们假设每台机器都有自己的内存和磁盘，一台机器不能访问另一台机器的内存或磁盘（除了通过网络向服务器发出请求）。

无共享并不是构建系统的唯一方式，但它已经成为构建互联网服务的主要方式，其原因如下：相对便宜，因为它不需要特殊的硬件，可以利用商品化的云计算服务，通过跨多个地理分布的数据中心进行冗余可以实现高可靠性。

互联网和数据中心（通常是以太网）中的大多数内部网络都是异步分组网络（**asynchronous packet networks**）。在这种网络中，一个节点可以向另一个节点发送一个消息（一个数据包），但是网络不能保证它什么时候到达，或者是否到达。如果您发送请求并期待响应，则很多事情可能会出错（其中一些如图8-1所示）：

1. 请求可能已经丢失（可能有人拔掉了网线）。
2. 请求可能正在排队，稍后将交付（也许网络或收件人超载）。
3. 远程节点可能已经失效（可能是崩溃或关机）。
4. 远程节点可能暂时停止了响应（可能会遇到长时间的垃圾回收暂停；参阅“暂停进程”），但稍后会再次响应。
5. 远程节点可能已经处理了请求，但是网络上的响应已经丢失（可能是网络交换机配置错误）。
6. 远程节点可能已经处理了请求，但是响应已经被延迟，并且稍后将被传递（可能是网络或者你自己的机器过载）。

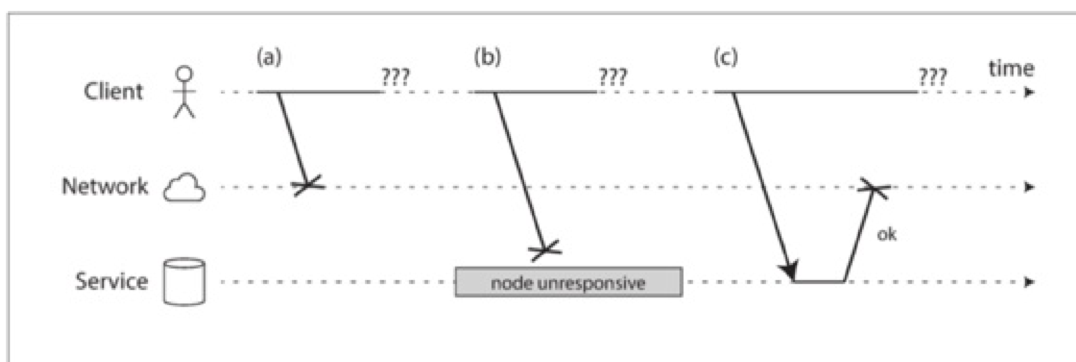


图**8-1** 如果发送请求并没有得到响应，则无法区分（**a**）请求是否丢失，（**b**）远程节点是否关闭，或（**c**）响应是否丢失。

发送者甚至不能分辨数据包是否被发送：唯一的选择是让接收者发送响应消息，这可能会丢失或延迟。这些问题在异步网络中难以区分：您所拥有的唯一信息是，您尚未收到响应。如果您向另一个节点发送请求并且没有收到响应，则无法说明原因。

处理这个问题的通常方法是超时（**Timeout**）：在一段时间之后放弃等待，并且认为响应不会到达。但是，当发生超时时，你仍然不知道远程节点是否收到了请求（如果请求仍然在某个地方排队，那么即使发件人已经放弃了该请求，仍然可能会将其发送给收件人）。

真实世界的网络故障

我们几十年来一直在建设计算机网络——有人可能希望现在我们已经找出了使网络变得可靠的方法。但是现在似乎还没有成功。 有一些系统的研究和大量的轶事证据表明，即使在像一家公司运营的数据中心那样的受控环境中，网络问题也可能出乎意料地普遍。在一家中型数据中心进行的一项研究发现，每个月大约有12个网络故障，其中一半断开一台机器，一半断开整个机架【15】。另一项研究测量了架顶式交换机，汇聚交换机和负载平衡器等组件的故障率【16】。它发现添加冗余网络设备不会像您所希望的那样减少故障，因为它不能防范人为错误（例如，错误配置的交换机），这是造成中断的主要原因。

诸如EC2之类的公有云服务因频繁的暂态网络故障而臭名昭着【14】，管理良好的私有数据中心网络可能是更稳定的环境。尽管如此，没有人不受网络问题的困扰：例如，交换机软件升级过程中的一个问题可能会引发网络拓扑重构，在此期间网络数据包可能会延迟超过一分钟【17】。鲨鱼可能咬住海底电缆并损坏它们 【18】。其他令人惊讶的故障包括网络接口有时会丢弃所有入站数据包，但是成功发送出站数据包 【19】：仅仅因为网络链接在一个方向上工作，并不能保证它也在相反的方向工作。

> 网络分区
>
> 当网络的一部分由于网络故障而被切断时，有时称为网络分区（**network partition**）或网络断裂（**netsplit**）。在本书中，我们通常会坚持使用更一般的术语网络故障（**network fault**），以避免与第6章讨论的存储系统的分区（分片）相混淆。

即使网络故障在你的环境中非常罕见，故障可能发生的事实，意味着你的软件需要能够处理它们。无论何时通过网络进行通信，都可能会失败，这是无法避免的。

如果网络故障的错误处理没有定义与测试，武断地讲，各种错误可能都会发生：例如，即使网络恢复【20】，集群可能会发生死锁，永久无法为请求提供服务，甚至可能会删除所有的数据【21】。如果软件被置于意料之外的情况下，它可能会做出出乎意料的事情。

处理网络故障并不意味着容忍它们：如果你的网络通常是相当可靠的，一个有效的方法可能是当你的网络遇到问题时，简单地向用户显示一条错误信息。但是，您确实需要知道您的软件如何应对网络问题，并确保系统能够从中恢复。有意识地触发网络问题并测试系统响应（这是Chaos Monkey背后的想法；参阅“可靠性”）。

检测故障

许多系统需要自动检测故障节点。例如：

- 负载平衡器需要停止向已死亡的节点转发请求（即从移出轮询列表（**out of rotation**））。
- 在单主复制功能的分布式数据库中，如果主库失效，则需要将从库之一升级为新主库（参阅“ch5.md#处理节点宕机”）。

不幸的是，网络的不确定性使得很难判断一个节点是否工作。在某些特定的情况下，您可能会收到一些反馈信息，明确告诉您某些事情没有成功：

- 如果你可以到达运行节点的机器，但没有进程正在侦听目标端口（例如，因为进程崩溃），操作系统将通过发送FIN或RST来关闭并重用TCP连接。但是，如果节点在处理请求时发生崩溃，则无法知道远程节点实际处理了多少数据【22】。
- 如果节点进程崩溃（或被管理员杀死），但节点的操作系统仍在运行，则脚本可以通知其他节点有关该崩溃的信息，以便另一个节点可以快速接管，而无需等待超时到期。例如，HBase做这个【23】。
- 如果您有权访问数据中心网络交换机的管理界面，则可以查询它们以检测硬件级别的链路故障（例如，远程机器是否关闭电源）。如果您通过互联网连接，或者如果您处于共享数据中心而无法访问交换机，或者由于网络问题而无法访问管理界面，则排除此选项。
- 如果路由器确认您尝试连接的IP地址不可用，则可能会使用ICMP目标不可达数据包回复您。但是，路由器不具备神奇的故障检测能力——它受到与网络其他参与者相同的限制。

关于远程节点关闭的快速反馈很有用，但是你不能指望它。即使TCP确认已经传送了一个数据包，应用程序在处理之前可能已经崩溃。如果你想确保一个请求是成功的，你需要应用程序本身的积极响应【24】。

相反，如果出了什么问题，你可能会在堆栈的某个层次上得到一个错误响应，但总的来说，你必须假设你根本就没有得到任何回应。您可以重试几次（TCP重试是透明的，但是您也可以在应用程序级别重试），等待超时过期，并且如果在超时时间内没有收到响应，则最终声明节点已经死亡。

超时与无穷的延迟

如果超时是检测故障的唯一可靠方法，那么超时应该等待多久？不幸的是没有简单的答案。

长时间的超时意味着长时间等待，直到一个节点被宣告死亡（在这段时间内，用户可能不得不等待，或者看到错误信息）。短暂的超时可以更快地检测到故障，但是实际上它只是经历了暂时的减速（例如，由于节点或网络上的负载峰值）而导致错误地宣布节点失效的风险更高。

过早地声明一个节点已经死了是有问题的：如果这个节点实际上是活着的，并且正在执行一些动作（例如，发送一封电子邮件），而另一个节点接管，那么这个动作可能会最终执行两次。我们将在“知识，真相和谎言”以及第9章和第11章中更详细地讨论这个问题。

当一个节点被宣告死亡时，它的职责需要转移到其他节点，这会给其他节点和网络带来额外的负担。如果系统已经处于高负荷状态，则过早宣告节点死亡会使问题更严重。尤其是可能发生，节点实际上并没有死亡，而是由于过载导致响应缓慢；将其负载转移到其他节点可能会导致级联失效（**cascading failure**）（在极端情况下，所有节点都宣告对方死亡，并且所有节点都停止工作）。

设想一个虚构的系统，其网络可以保证数据包的最大延迟——每个数据包要么在一段时间内传送，要么丢失，但是传递永远不会比d更长。此外，假设你可以保证一个非故障节点总是在一段时间内处理一个请求r。在这种情况下，您可以保证每个成功的请求在$2d + r$时间内都能收到响应，如果您在此时间内没有收到响应，则知道网络或远程节点不工作。如果这是成立的，$2d + r$ 会是一个合理的超时设置。

不幸的是，我们所使用的大多数系统都没有这些保证：异步网络具有无限的延迟（即尽可能快地传送数据包，但数据包到达可能需要的时间没有上限），并且大多数服务器实现并不能保证它们可以在一定的最大时间内处理请求（请参阅“响应时间保证”）。对于故障检测，系统大部分时间快速运行是不够的：如果你的超时时间很短，往返时间只需要一个瞬时尖峰就可以使系统失衡。

网络拥塞和排队

在驾驶汽车时，由于交通拥堵，道路交通网络的通行时间往往不尽相同。同样，计算机网络上数据包延迟的可变性通常是由于排队【25】：

- 如果多个不同的节点同时尝试将数据包发送到同一目的地，则网络交换机必须将它们排队并将它们逐个送入目标网络链路（如图8-2所示）。在繁忙的网络链路上，数据包可能需要等待一段时间才能获得一个插槽（这称为网络连接）。如果传入的数据太多，交换机队列填满，数据包将被丢弃，因此需要重新发送数据包 - 即使网络运行良好。
- 当数据包到达目标机器时，如果所有CPU内核当前都处于繁忙状态，则来自网络的传入请求将被操作系统排队，直到应用程序准备好处理它为止。根据机器上的负载，这可能需要一段任意的时间。
- 在虚拟化环境中，正在运行的操作系统经常暂停几十毫秒，而另一个虚拟机使用CPU内核。在这段时间内，虚拟机不能从网络中消耗任何数据，所以传入的数据被虚拟机监视器 【26】排队（缓冲），进一步增加了网络延迟的可变性。
- TCP执行流量控制（**flow control**）（也称为拥塞避免（**congestion avoidance**）或背压（**backpressure**）），其中节点限制自己的发送速率以避免网络链路或接收节点过载【27】。这意味着在数据甚至进入网络之前，在发送者处需要进行额外的排队。

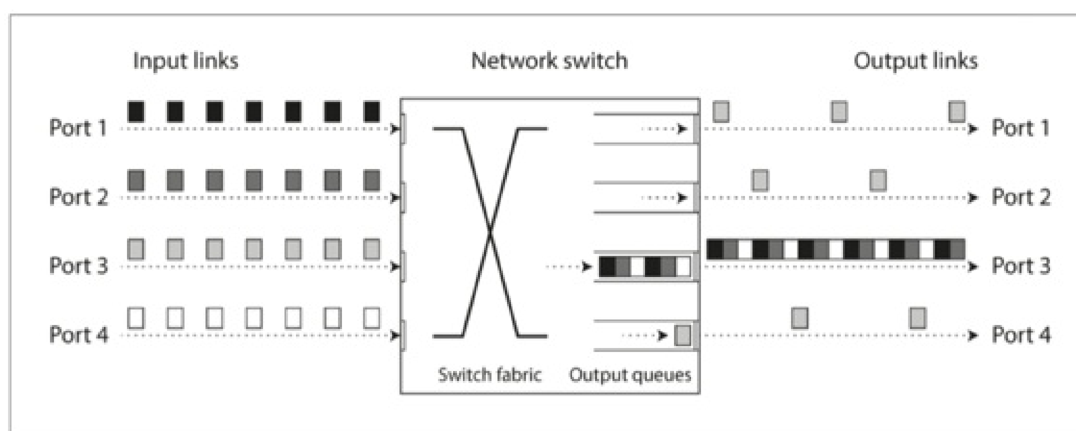


图**8-2** 如果有多台机器将网络流量发送到同一目的地，则其交换机队列可能会被填满。在这里，端口**1,2**和**4**都试图发送数据包到端口**3**

而且，如果TCP在某个超时时间内没有被确认（这是根据观察的往返时间计算的），则认为数据包丢失，丢失的数据包将自动重新发送。尽管应用程序没有看到数据包丢失和重新传输，但它看到了延迟（等待超时到期，然后等待重新传输的数据包得到确认）。

> ## TCP与UDP
>
> 一些对延迟敏感的应用程序（如视频会议和IP语音（VoIP））使用UDP而不是TCP。这是在可靠性和和延迟可变性之间的折衷：由于UDP不执行流量控制并且不重传丢失的分组，所以避免了可变网络延迟的一些原因（尽管它仍然易受切换队列和调度延迟的影响）。
>
> 在延迟数据毫无价值的情况下，UDP是一个不错的选择。例如，在VoIP电话呼叫中，可能没有足够的时间重新发送丢失的数据包，并在扬声器上播放数据。在这种情况下，重发数据包没有意义——应用程序必须使用静音填充丢失数据包的时隙（导致声音短暂中断），然后在数据流中继续。重试发生在人类层。（“你能再说一遍吗？声音刚刚断了一会儿。“）

所有这些因素都会造成网络延迟的变化。当系统接近其最大容量时，排队延迟的范围特别广泛：

拥有足够备用容量的系统可以轻松排空队列，而在高利用率的系统中，很快就能积累很长的队列。

在公共云和多租户数据中心中，资源被许多客户共享：网络链接和交换机，甚至每个机器的网卡和CPU（在虚拟机上运行时）。批处理工作负载（如MapReduce）（参阅第10章）可能很容易使网络链接饱和。由于无法控制或了解其他客户对共享资源的使用情况，如果附近的某个人（嘈杂的邻居）正在使用大量资源，则网络延迟可能会发生剧烈抖动【28,29】。

在这种环境下，您只能通过实验方式选择超时：测量延长的网络往返时间和多台机器的分布，以确定延迟的预期可变性。然后，考虑到应用程序的特性，可以确定故障检测延迟与过早超时风险之间的适当折衷。

更好的一种做法是，系统不是使用配置的常量超时，而是连续测量响应时间及其变化（抖动），并根据观察到的响应时间分布自动调整超时。这可以通过Phi Accrual故障检测器【30】来完成，该检测器例如在Akka和Cassandra 【31】中使用。 TCP重传超时也同样起作用【27】。

同步网络 vs 异步网络

如果我们可以依靠网络来传递一些最大延迟固定的数据包，而不是丢弃数据包，那么分布式系统就会简单得多。为什么我们不能在硬件层面上解决这个问题，使网络可靠，使软件不必担心呢？

为了回答这个问题，将数据中心网络与非常可靠的传统固定电话网络（非蜂窝，非VoIP）进行比较是很有趣的：延迟音频帧和掉话是非常罕见的。一个电话需要一个很低的端到端延迟，以及足够的带宽来传输你声音的音频采样数据。在计算机网络中有类似的可靠性和可预测性不是很好吗？

当您通过电话网络拨打电话时，它会建立一个电路：在两个呼叫者之间的整个路线上为呼叫分配一个固定的，有保证的带宽量。这个电路会保持至通话结束【32】。例如，ISDN网络以每秒4000帧的固定速率运行。呼叫建立时，每个帧内（每个方向）分配16位空间。因此，在通话期间，每一方都保证能够每250微秒发送一个精确的16位音频数据【33,34】。

这种网络是同步的：即使数据经过多个路由器，也不会受到排队的影响，因为呼叫的16位空间已经在网络的下一跳中保留了下来。而且由于没有排队，网络的最大端到端延迟是固定的。我们称之为有限延迟（**bounded delay**）。

我们不能简单地使网络延迟可预测吗？

请注意，电话网络中的电路与TCP连接有很大不同：电路是固定数量的预留带宽，在电路建立时没有其他人可以使用，而TCP连接的数据包机会性地使用任何可用的网络带宽。您可以给TCP一个可变大小的数据块（例如，一个电子邮件或一个网页），它会尽可能在最短的时间内传输它。 TCP连接空闲时，不使用任何带宽[ii]。

> [ii]. 除了偶尔的keepalive数据包，如果TCP keepalive被启用。 ↩

如果数据中心网络和互联网是电路交换网络，那么在建立电路时就可以建立一个保证的最大往返时间。但是，它们并不是：以太网和IP是分组交换协议，这些协议可以从排队中获得，从而使网络无限延迟。这些协议没有电路的概念。

为什么数据中心网络和互联网使用分组交换？答案是，它们针对突发流量（**bursty traffic**）进行了优化。一个电路适用于音频或视频通话，在通话期间需要每秒传送相当数量的比特。另一方面，请求网页，发送电子邮件或传输文件没有任何特定的带宽要求——我们只是希望它尽快完成。

如果你想通过电路传输文件，你将不得不猜测一个带宽分配。如果您猜的太低，传输速度会不必要的太慢，导致网络容量不能使用。如果你猜的太高，电路就无法建立（因为如果无法保证其带宽分配，网络不能建立电路）。因此，使用用于突发数据传输的电路浪费网络容量，并且使传输不必要地缓慢。相比之下，TCP动态调整数据传输速率以适应可用的网络容量。

已经有一些尝试去建立支持电路交换和分组交换的混合网络，比如ATM[iii] InfiniBand有一些相似之处【35】：它在链路层实现了端到端的流量控制，从而减少了在网络中排队，尽管它仍然可能因链路拥塞而受到延迟【36】。通过仔细使用服务质量（**quality of service,**）（QoS，数据包的优先级和调度）和准入控制（**admission control**）（限速发送器），可以仿真分组网络上的电路交换，或提供统计上的有限延迟【25,32】。

> iii. 异步传输模式（**Asynchronous TransferMode, ATM**）在20世纪80年代是以太网的竞争对手【32】，但在电话网核心交换机之外并没有得到太多的采用。与自动柜员机（也称为自动取款机）无关，尽管共用一个缩写词。或许，在一些平行的世界里，互联网是基于像ATM这样的东西，因为互联网视频通话可能比我们的更可靠，因为它们不会遭受丢包和延迟的包裹。↩

但是，目前在多租户数据中心和公共云或通过互联网[iv]进行通信时，此类服务质量尚未启用。当前部署的技术不允许我们对网络的延迟或可靠性作出任何保证：我们必须假设网络拥塞，排队和无限的延迟总是会发生。 因此，超时时间没有“正确”的值——它需要通过实验来确定。

> iv. 互联网服务提供商之间的对等协议和通过**BGP**网关协议（**BGP**）建立路由之间的对等协议，与电路交换本身相比，与电路交换更接近。在这个级别上，可以购买专用带宽。但是，互联网路由在网络级别运行，而不是主机之间的单独连接，而且运行时间要长得多。↩

> ### 延迟和资源利用
>
> 更一般地说，可以将延迟变化视为动态资源分区的结果。
>
> 假设两台电话交换机之间有一条线路，可以同时进行10,000个呼叫。通过此线路切换的每个电路都占用其中一个呼叫插槽。因此，您可以将线路视为可由多达10,000个并发用户共享的资源。资源以静态方式分配：即使您现在是电话上唯一的电话，并且所有其他9,999个插槽都未使用，您的电路仍将分配与导线充分利用时相同的固定数量的带宽。
>
> 相比之下，互联网动态分享网络带宽。发送者互相推挤并互相推挤以尽可能快地通过网络获得它们的分组，并且网络交换机决定从一个时刻到另一个时刻发送哪个分组（即，带宽分配）。这种方法有排队的缺点，但其优点是它最大限度地利用了电线。电线固定成本，所以如果你更好地利用它，你通过电线发送的每个字节都会更便宜。
>
> CPU也会出现类似的情况：如果您在多个线程间动态共享每个CPU内核，则有一个线程有时必须等待操作系统的运行队列，而另一个线程正在运行，这样线程可以暂停不同的时间长度。但是，与为每个线程分配静态数量的CPU周期相比，这会更好地利用硬件（参阅“响应时间保证”）。更好的硬件利用率也是使用虚拟机的重要动机。
>
> 如果资源是静态分区的（例如，专用硬件和专用带宽分配），则在某些环境中可以实现延迟保证。但是，这是以降低利用率为代价的——换句话说，它是更昂贵的。另一方面，动态资源分配的多租户提供了更好的利用率，所以它更便宜，但它具有可变延迟的缺点。
>
> 网络中的可变延迟不是一种自然规律，而只是成本/收益权衡的结果。

不可靠的时钟

时钟和时间很重要。应用程序以各种方式依赖于时钟来回答以下问题：

1. 这个请求是否超时了？
2. 这项服务的第99百分位响应时间是多少？
3. 在过去五分钟内，该服务平均每秒处理多少个查询？
4. 用户在我们的网站上花了多长时间？
5. 这篇文章在何时发布？
6. 在什么时间发送提醒邮件？
7. 这个缓存条目何时到期？
8. 日志文件中此错误消息的时间戳是什么？

例1-4测量持续时间（例如，发送请求与正在接收的响应之间的时间间隔），而例5-8描述时间点（**point in time**）（在特定日期，特定时间发生的事件）。

在分布式系统中，时间是一件棘手的事情，因为通信不是即时的：消息通过网络从一台机器传送到另一台机器需要时间。收到消息的时间总是晚于发送的时间，但是由于网络中的可变延迟，我们不知道多少时间。这个事实有时很难确定在涉及多台机器时发生事情的顺序。

而且，网络上的每台机器都有自己的时钟，这是一个实际的硬件设备：通常是石英晶体振荡器。这些设备不是完全准确的，所以每台机器都有自己的时间概念，可能比其他机器稍快或更慢。可以在一定程度上同步时钟：最常用的机制是网络时间协议（**NTP**），它允许根据一组服务器报告的时间来调整计算机时钟【37】。服务器则从更精确的时间源（如GPS接收机）获取时间。

单调钟与时钟

现代计算机至少有两种不同的时钟：时钟和单调钟。尽管它们都衡量时间，但区分这两者很重要，因为它们有不同的目的。

时钟

时钟是您直观地了解时钟的依据：它根据某个日历（也称为挂钟时间（**wall-clock time**））返回当前日期和时间。例如，Linux[V]上的 `clock_gettime(CLOCK_REALTIME)` 和Java中的 `System.currentTimeMillis()` 返回自epoch（1970年1月1日 午夜 UTC，格里高利历）以来的秒数（或毫秒），根据公历日历，不包括闰秒。有些系统使用其他日期作为参考点。

> V. 虽然时钟被称为实时时钟，但它与实时操作系统无关，如第298页上的“响应时间保证”中所述。↩

时钟通常与NTP同步，这意味着来自一台机器的时间戳（理想情况下）意味着与另一台机器上的时间戳相同。但是如下节所述，时钟也具有各种各样的奇特之处。特别是，如果本地时钟在NTP服务器之前太远，则它可能会被强制重置，看上去好像跳回了先前的时间点。这些跳跃以及他们经常忽略闰秒的事实，使时钟不能用于测量经过时间【38】。

时钟还具有相当粗略的分辨率，例如，在较早的Windows系统上以10毫秒为单位前进【39】。在最近的系统中这已经不是一个问题了。

单调钟

单调钟适用于测量持续时间（时间间隔），例如超时或服务的响应时间：Linux上的 `clock_gettime(CLOCK_MONOTONIC)` ，和Java中的 `System.nanoTime()` 都是单调时钟。这个名字来源于他们保证总是前进的事实（而时钟可以及时跳回）。

时钟同步与准确性

单调钟不需要同步，但是时钟需要根据NTP服务器或其他外部时间源来设置才能有用。不幸的是，我们获取时钟的方法并不像你所希望的那样可靠或准确——硬件时钟和NTP可能会变幻莫测。举几个例子：

计算机中的石英钟不够精确：它会漂移（**drifts**）（运行速度快于或慢于预期）。时钟漂移取决于机器的温度。 Google假设其服务器时钟漂移为200 ppm（百万分之一）【41】，相当于每30秒与服务器重新同步一次的时钟漂移为6毫秒，或者每天重新同步的时钟漂移为17秒。即使一切工作正常，此漂移也会限制可以达到的最佳准确度。

- 如果计算机的时钟与NTP服务器的时钟差别太大，可能会拒绝同步，或者本地时钟将被强制重置【37】。任何观察重置前后时间的应用程序都可能会看到时间倒退或突然跳跃。
- 如果某个节点被NTP服务器意外阻塞，可能会在一段时间内忽略错误配置。有证据表明，这在实践中确实发生过。
- NTP同步只能和网络延迟一样好，所以当您在拥有可变数据包延迟的拥塞网络上时，NTP同步的准确性会受到限制。一个实验表明，当通过互联网同步时，35毫秒的最小误差是可以实现的，尽管偶尔的网络延迟峰值会导致大约一秒的误差。根据配置，较大的网络延迟会导致NTP客户端完全放弃。
- 一些NTP服务器错误或配置错误，报告时间已经过去了几个小时【43,44】。 NTP客户端非常强大，因为他们查询多个服务器并忽略异常值。尽管如此，在互联网上陌生人告诉你的时候，你的系统的正确性还是值得担忧的。
- 闰秒导致59分钟或61秒长的分钟，这混淆了未设计闰秒的系统中的时序假设【45】。闰秒已经使许多大型系统崩溃【38,46】的事实说明了，关于时钟的假设是多么容易偷偷溜入系统中。处理闰秒的最佳方法可能是通过在一天中逐渐执行闰秒调整（这被称为拖尾（**smearing**））【47,48】，使NTP服务器“撒谎”，虽然实际的NTP服务器表现各异【49】。
- 在虚拟机中，硬件时钟被虚拟化，这对于需要精确计时的应用程序提出了额外的挑战【50】。当一个CPU核心在虚拟机之间共享时，每个虚拟机都会暂停几十毫秒，而另一个虚拟机正在运行。从应用程序的角度来看，这种停顿表现为时钟突然向前跳跃【26】。
- 如果您在未完全控制的设备上运行软件（例如，移动设备或嵌入式设备），则可能完全不信任该设备的硬件时钟。一些用户故意将其硬件时钟设置为不正确的日期和时间，例如，为了规避游戏中的时间限制，时钟可能会被设置到很远的过去或将来。

如果你足够关心这件事并投入大量资源，就可以达到非常好的时钟精度。例如，针对金融机构的欧洲法规草案MiFID II要求所有高频率交易基金在UTC时间100微秒内同步时钟，以便调试“闪崩”等市场异常现象，并帮助检测市场操纵 【51】。

使用GPS接收机，精确时间协议（PTP）【52】以及仔细的部署和监测可以实现这种精确度。然而，这需要很多努力和专业知识，而且有很多东西都会导致时钟同步错误。如果你的NTP守护进程配置错误，或者防火墙阻止了NTP通信，由漂移引起的时钟误差可能很快就会变大。

依赖同步时钟

时钟的问题在于，虽然它们看起来简单易用，但却具有令人惊讶的缺陷：一天可能不会有精确的86,400秒，时钟可能会前后跳跃，而一个节点上的时间可能与另一个节点上的时间完全不同。

本章早些时候，我们讨论了网络丢包和任意延迟包的问题。尽管网络在大多数情况下表现良好，但软件的设计必须假定网络偶尔会出现故障，而软件必须正常处理这些故障。时钟也是如此：尽管大多数时间都工作得很好，但需要准备健壮的软件来处理不正确的时钟。

有一部分问题是，不正确的时钟很容易被视而不见。如果一台机器的CPU出现故障或者网络配置错误，很可能根本无法工作，所以很快就会被注意和修复。另一方面，如果它的石英时钟有缺陷，或者它的NTP客户端配置错误，大部分事情似乎仍然可以正常工作，即使它的时钟逐渐偏离现实。如果某个软件依赖于精确同步的时钟，那么结果更可能是悄无声息且行踪渺茫数据的数据丢失，而不是一次惊天动地的崩溃【53,54】。

因此，如果你使用需要同步时钟的软件，必须仔细监控所有机器之间的时钟偏移。时钟偏离其他时钟太远的节点应当被宣告死亡，并从集群中移除。这样的监控可以确保你在损失发生之前注意到破损的时钟。

有序事件的时间戳

让我们考虑一个特别的情况，一件很有诱惑但也很危险的事情：依赖时钟，在多个节点上对事件进行排序。 例如，如果两个客户端写入分布式数据库，谁先到达？ 哪一个更近？

图8-3显示了在具有多领导者复制的数据库中对时钟的危险使用（该例子类似于图5-9）。 客户端A在节点1上写入 `x = 1` ；写入被复制到节点3；客户端B在节点3上增加x（我们现在有 `x = 2` ）；最后这两个写入都被复制到节点2。

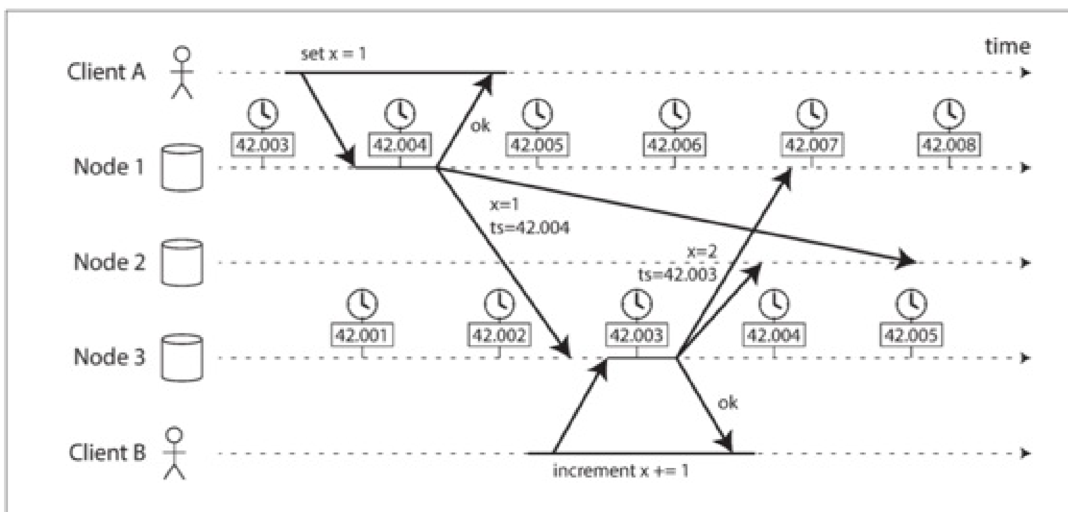


图**8-3** 客户端**B**的写入比客户端**A**的写入要晚，但是**B**的写入具有较早的时间戳。

在图8-3中，当一个写入被复制到其他节点时，它会根据发生写入的节点上的时钟时钟标记一个时间戳。在这个例子中，时钟同步是非常好的：节点1和节点3之间的偏差小于3ms，这可能比你在实践中预期的更好。

尽管如此，图8-3中的时间戳却无法正确排列事件：写入 `x = 1` 的时间戳为42.004秒，但写入 `x = 2` 的时间戳为42.003秒，即使 `x = 2` 在稍后出现。当节点2接收到这两个事件时，会错误地推断出 `x = 1` 是最近的值，而写入 `x = 2` 。效果上表现为，客户端B的增量操作会丢失。

这种冲突解决策略被称为最后写入为准（**LWW**），它在多领导者复制和无领导者数据库（如Cassandra 【53】和Riak 【54】）中被广泛使用（参见“最后写入为准（丢弃并发写入）”一节）。有些实现会在客户端而不是服务器上生成时间戳，但这并不能改变LWW的基本问题：

- 数据库写入可能会神秘地消失：具有滞后时钟的节点无法用快速时钟覆盖之前由节点写入的值，直到节点之间的时钟偏差过去【54,55】。此方案可能导致一定数量的数据被悄悄丢弃，而未向应用报告任何错误。
- LWW无法区分高频顺序写入（在图8-3中，客户端B的增量操作一定发生在客户端A的写入之后）和真正并发写入（写入者意识不到其他写入者）。需要额外的因果关系跟踪机制（例如版本向量），以防止因果关系的冲突（请参阅“检测并发写入”）。
- 两个节点可以独立生成具有相同时间戳的写入，特别是在时钟仅具有毫秒分辨率的情况下。为了解决这样的冲突，还需要一个额外的决胜值（**tiebreaker**）（可以简单地是一个大随机数），但这种方法也可能会导致违背因果关系【53】。

因此，尽管通过保留最“最近”的值并放弃其他值来解决冲突是很诱惑人的，但是要注意，“最近”的定义取决于本地的时钟，这很可能是不正确的。即使用频繁同步的NTP时钟，一个数据包也可能在时间戳100毫秒（根据发送者的时钟）时发送，并在时间戳99毫秒（根据接收者的时钟）处到达——看起来好像数据包在发送之前已经到达，这是不可能的。

NTP同步是否能足够准确，以至于这种不正确的排序不会发生？也许不能，因为NTP的同步精度本身受到网络往返时间的限制，除了石英钟漂移这类误差源之外。为了进行正确的排序，你需要一个比测量对象（即网络延迟）要精确得多的时钟。

所谓的逻辑时钟【56,57】是基于递增计数器而不是振荡石英晶体，对于排序事件来说是更安全的选择（请参见“检测并发写入”）。逻辑时钟不测量一天中的时间或经过的秒数，而仅测量事件的相对顺序（无论一个事件发生在另一个事件之前还是之后）。相反，用来测量实际经过时间的时钟和单调钟也被称为物理时钟。我们将在“顺序保证”中查看更多订购信息。

时钟读数存在置信区间

您可能能够以微秒或甚至纳秒的分辨率读取机器的时钟。但即使可以得到如此细致的测量结果，这并不意味着这个值对于这样的精度实际上是准确的。实际上，如前所述，即使您每分钟与本地网络上的NTP服务器进行同步，很可能也不会像前面提到的那样，在不精确的石英时钟上漂移几毫秒。使用公共互联网上的NTP服务器，最好的准确度可能达到几十毫秒，而且当网络拥塞时，误差可能会超过100毫秒【57】。

因此，将时钟读数视为一个时间点是没有意义的——它更像是一段时间范围：例如，一个系统可能以95%的置信度认为当前时间处于本分钟内的第10.3秒和10.5秒之间，它可能没法比这更精确了【58】。如果我们只知道±100毫秒的时间，那么时间戳中的微秒数字部分基本上

是没有意义的。

不确定性界限可以根据你的时间源来计算。如果您的GPS接收器或原子（铯）时钟直接连接到您的计算机上，预期的错误范围由制造商报告。如果从服务器获得时间，则不确定性取决于自上次与服务器同步以来的石英钟漂移的期望值，加上NTP服务器的不确定性，再加上到服务器的网络往返时间（只是获取粗略近似值，并假设服务器是可信的）。

不幸的是，大多数系统不公开这种不确定性：例如，当调用 `clock_gettime()` 时，返回值不会告诉你时间戳的预期错误，所以你不知道其置信区间是5毫秒还是5年。

一个有趣的例外是Spanner中的Google TrueTime API 【41】，它明确地报告了本地时钟的置信区间。当你询问当前时间时，你会得到两个值：[最早，最晚]，这是最早可能的时间戳和最晚可能的时间戳。在不确定性估计的基础上，时钟知道当前的实际时间落在该区间内。间隔的宽度取决于自从本地石英钟最后与更精确的时钟源同步以来已经过了多长时间。

全局快照的同步时钟

在“快照隔离和可重复读取”中，我们讨论了快照隔离，这是数据库中非常有用的功能，需要支持小型快速读写事务和大型长时间运行的只读事务，用于备份或分析）。它允许只读事务看到特定时间点的处于一致状态的数据库，且不会锁定和干扰读写事务。

快照隔离最常见的实现需要单调递增的事务ID。如果写入比快照晚（即，写入具有比快照更大的事务ID），则该写入对于快照事务是不可见的。在单节点数据库上，一个简单的计数器就足以生成事务ID。

但是当数据库分布在许多机器上，也许可能在多个数据中心中时，由于需要协调，（跨所有分区）全局单调递增的事务ID可能很难生成。事务ID必须反映因果关系：如果事务B读取由事务A写入的值，则B必须具有比A更大的事务ID，否则快照就无法保持一致。在有大量的小规模、高频率的事务情景下，在分布式系统中创建事务ID成为一个站不住脚的瓶颈[vi]。

> [vi]. 存在分布式序列号生成器，例如Twitter的雪花（Snowflake），其以可扩展的方式（例如，通过将ID空间的块分配给不同节点）近似单调地增加唯一ID。但是，它们通常无法保证与因果关系一致的排序，因为分配的ID块的时间范围比数据库读取和写入的时间范围要长。另请参阅“顺序保证”。↩

我们可以使用同步时钟的时间戳作为事务ID吗？如果我们能够获得足够好的同步性，那么这种方法将具有很合适的属性：更晚的事务会有更大的时间戳。当然，问题在于时钟精度的不确定性。

Spanner以这种方式实现跨数据中心的快照隔离【59，60】。它使用TrueTime API报告的时钟置信区间，并基于以下观察结果：如果您有两个置信区间，每个置信区间包含最早和最近可能的时间戳（ $A = [A_{earliest}, A_{latest}]$， $B=[B_{earliest}, B_{latest}]$），这两个区间不重叠（即：$A_{earliest} < A_{latest} < B_{earliest} < B_{latest}$），那么B肯定发生在A之后——这是毫无疑问的。只有当区间重叠时，我们才不确定A和B发生的顺序。

为了确保事务时间戳反映因果关系，在提交读写事务之前，Spanner在提交读写事务时，会故意等待置信区间长度的时间。通过这样，它可以确保任何可能读取数据的事务处于足够晚的时间，因此它们的置信区间不会重叠。为了保持尽可能短的等待时间，Spanner需要保持尽可能小的时钟不确定性，为此，Google在每个数据中心都部署了一个GPS接收器或原子钟，允许时钟在大约7毫秒内同步【41】。

对分布式事务语义使用时钟同步是一个活跃的研究领域【57,61,62】。这些想法很有趣，但是它们还没有在谷歌之外的主流数据库中实现。

暂停进程

让我们考虑在分布式系统中使用危险时钟的另一个例子。假设你有一个数据库，每个分区只有一个领导者。只有领导被允许接受写入。一个节点如何知道它仍然是领导者（它并没有被别人宣告为死亡），并且它可以安全地接受写入？

一种选择是领导者从其他节点获得一个租约（**lease**），类似一个带超时的锁【63】。任一时刻只有一个节点可以持有租约——因此，当一个节点获得一个租约时，它知道它在某段时间内自己是领导者，直到租约到期。为了保持领导地位，节点必须在周期性地在租约过期前续期。

如果节点发生故障，就会停止续期，所以当租约过期时，另一个节点可以接管。

可以想象，请求处理循环看起来像这样：

```
while(true){
    request=getIncomingRequest();
    // 确保租约还剩下至少10秒
    if (lease.expiryTimeMillis-System.currentTimeMillis()< 10000){
        lease = lease.renew();
    }

    if(lease.isValid()){
        process(request);
    }}
}
```

这个代码有什么问题？首先，它依赖于同步时钟：租约到期时间由另一台机器设置（例如，当前时间加上30秒，计算到期时间），并将其与本地系统时钟进行比较。如果时钟超过几秒不同步，这段代码将开始做奇怪的事情。

其次，即使我们将协议更改为仅使用本地单调时钟，也存在另一个问题：代码假定在执行剩余时间检查 `System.currentTimeMillis()` 和实际执行请求 `process(request)` 中间的时间间隔非常短。通常情况下，这段代码运行得非常快，所以10秒的缓冲区已经足够确保租约在请求处理到一半时不会过期。

但是，如果程序执行中出现了意外的停顿呢？例如，想象一下，线程在 `lease.isValid()` 行周围停止15秒，然后才终止。在这种情况下，在请求被处理的时候，租约可能已经过期，而另一个节点已经接管了领导。然而，没有什么可以告诉这个线程已经暂停了这么长时间了，所

以这段代码不会注意到租约已经到期了，直到循环的下一个迭代 ——到那个时候它可能已经做了一些不安全的处理请求。

假设一个线程可能会暂停很长时间，这是疯了吗？不幸的是，这种情况发生的原因有很多种：

- 许多编程语言运行时（如Java虚拟机）都有一个垃圾收集器（GC），偶尔需要停止所有正在运行的线程。这些“停止世界（**stop-the-world**）”GC暂停有时会持续几分钟【64】！甚至像HotSpot JVM的CMS这样的所谓的“并行”垃圾收集器也不能完全与应用程序代码并行运行，它需要不时地停止世界【65】。尽管通常可以通过改变分配模式或调整GC设置来减少暂停【66】，但是如果我们想要提供健壮的保证，就必须假设最坏的情况发生。
- 在虚拟化环境中，可以挂起（**suspend**）虚拟机（暂停执行所有进程并将内存内容保存到磁盘）并恢复（恢复内存内容并继续执行）。这个暂停可以在进程执行的任何时候发生，并且可以持续任意长的时间。这个功能有时用于虚拟机从一个主机到另一个主机的实时迁移，而不需要重新启动，在这种情况下，暂停的长度取决于进程写入内存的速率【67】。
- 在最终用户的设备（如笔记本电脑）上，执行也可能被暂停并随意恢复，例如当用户关闭笔记本电脑的盖子时。
- 当操作系统上下文切换到另一个线程时，或者当管理程序切换到另一个虚拟机时（在虚拟机中运行时），当前正在运行的线程可以在代码中的任意点处暂停。在虚拟机的情况下，在其他虚拟机中花费的CPU时间被称为窃取时间（**steal time**）。如果机器处于沉重的负载下（即，如果等待运行的线程很长），暂停的线程再次运行可能需要一些时间。
- 如果应用程序执行同步磁盘访问，则线程可能暂停，等待缓慢的磁盘I/O操作完成【68】。在许多语言中，即使代码没有包含文件访问，磁盘访问也可能出乎意料地发生——例如，Java类加载器在第一次使用时惰性加载类文件，这可能在程序执行过程中随时发生。 I/O暂停和GC暂停甚至可能合谋组合它们的延迟【69】。如果磁盘实际上是一个网络文件系统或网络块设备（如亚马逊的EBS），I/O延迟进一步受到网络延迟变化的影响【29】。
- 如果操作系统配置为允许交换到磁盘（分页），则简单的内存访问可能导致页面错误（**page fault**），要求将磁盘中的页面装入内存。当这个缓慢的I/O操作发生时，线程暂停。如果内存压力很高，则可能需要将不同的页面换出到磁盘。在极端情况下，操作系统可能花费大部分时间将页面交换到内存中，而实际上完成的工作很少（这被称为抖动（**thrashing**））。为了避免这个问题，通常在服务器机器上禁用页面调度（如果你宁愿干掉一个进程来释放内存，也不愿意冒抖动风险）。
- 可以通过发送SIGSTOP信号来暂停Unix进程，例如通过在shell中按下Ctrl-Z。 这个信号立即阻止进程继续执行更多的CPU周期，直到SIGCONT恢复为止，此时它将继续运行。即使你的环境通常不使用SIGSTOP，也可能由运维工程师意外发送。

所有这些事件都可以随时抢占（**preempt**）正在运行的线程，并在稍后的时间恢复运行，而线程甚至不会注意到这一点。这个问题类似于在单个机器上使多线程代码线程安全：你不能对时机做任何假设，因为随时可能发生上下文切换，或者出现并行运行。

当在一台机器上编写多线程代码时，我们有相当好的工具来实现线程安全：互斥量，信号量，原子计数器，无锁数据结构，阻塞队列等等。不幸的是，这些工具并不能直接转化为分布式系统操作，因为分布式系统没有共享内存，只有通过不可靠网络发送的消息。

分布式系统中的节点，必须假定其执行可能在任意时刻暂停相当长的时间，即使是在一个函数的中间。在暂停期间，世界的其它部分在继续运转，甚至可能因为该节点没有响应，而宣告暂停节点的死亡。最终暂停的节点可能会继续运行，在再次检查自己的时钟之前，甚至可能不会意识到自己进入了睡眠。

响应时间保证

在许多编程语言和操作系统中，线程和进程可能暂停一段无限制的时间，正如讨论的那样。如果你足够努力，导致暂停的原因是可以消除的。

某些软件的运行环境要求很高，不能在特定时间内响应可能会导致严重的损失：飞机主控计算机，火箭，机器人，汽车和其他物体的计算机必须对其传感器输入做出快速而可预测的响应。在这些系统中，软件必须有一个特定的截止时间（**deadline**），如果截止时间不满足，可能会导致整个系统的故障。这就是所谓的硬实时（**hard real-time**）系统。

> #### 实时是真的吗？
>
> 在嵌入式系统中，实时是指系统经过精心设计和测试，以满足所有情况下的特定时间保证。这个含义与Web上实时术语的模糊使用相反，它描述了服务器将数据推送到客户端以及流处理，而没有严格的响应时间限制（见第11章）。

例如，如果车载传感器检测到当前正在经历碰撞，你肯定不希望安全气囊释放系统因为GC暂停而延迟弹出。

在系统中提供实时保证需要各级软件栈的支持：一个实时操作系统（RTOS），允许在指定的时间间隔内保证CPU时间的分配。库函数必须记录最坏情况下的执行时间；动态内存分配可能受到限制或完全不允许（实时垃圾收集器存在，但是应用程序仍然必须确保它不会给GC太多的负担）；必须进行大量的测试和测量，以确保达到保证。

所有这些都需要大量额外的工作，严重限制了可以使用的编程语言，库和工具的范围（因为大多数语言和工具不提供实时保证）。由于这些原因，开发实时系统非常昂贵，并且它们通常用于安全关键的嵌入式设备。而且，“实时”与“高性能”不一样——事实上，实时系统可能具有较低的吞吐量，因为他们必须优先考虑及时响应高于一切（另请参见“延迟和资源利用“）。

对于大多数服务器端数据处理系统来说，实时保证是不经济或不合适的。因此，这些系统必须承受在非实时环境中运行的暂停和时钟不稳定性。

限制垃圾收集的影响

过程暂停的负面影响可以在不诉诸昂贵的实时调度保证的情况下得到缓解。语言运行时在计划垃圾回收时具有一定的灵活性，因为它们可以跟踪对象分配的速度和随着时间的推移剩余的空闲内存。

一个新兴的想法是将GC暂停视为一个节点的短暂计划中断，并让其他节点处理来自客户端的请求，同时一个节点正在收集其垃圾。如果运行时可以警告应用程序一个节点很快需要GC暂停，那么应用程序可以停止向该节点发送新的请求，等待它完成处理未完成的请求，然后在没有请求正在进行时执行GC。这个技巧隐藏了来自客户端的GC暂停，并降低了响应时间的高百分比【70,71】。一些对延迟敏感的金融交易系统【72】使用这种方法。

这个想法的一个变种是只用垃圾收集器来处理短命对象（这些对象要快速收集），并定期在积累大量长寿对象（因此需要完整GC）之前重新启动进程【65,73】。一次可以重新启动一个节点，在计划重新启动之前，流量可以从节点移开，就像滚动升级一样。

这些措施不能完全阻止垃圾回收暂停，但可以有效地减少它们对应用的影响。

知识、真相与谎言

本章到目前为止，我们已经探索了分布式系统与运行在单台计算机上的程序的不同之处：没有共享内存，只有通过可变延迟的不可靠网络传递的消息，系统可能遭受部分失效，不可靠的时钟和处理暂停。

如果你不习惯于分布式系统，那么这些问题的后果就会让人迷惑不解。网络中的一个节点无法确切地知道任何事情——它只能根据它通过网络接收到（或没有接收到）的消息进行猜测。节点只能通过交换消息来找出另一个节点所处的状态（存储了哪些数据，是否正确运行等等）。如果远程节点没有响应，则无法知道它处于什么状态，因为网络中的问题不能可靠地与节点上的问题区分开来。

这些系统的讨论与哲学有关：在系统中什么是真什么是假？如果感知和测量的机制都是不可靠的，那么关于这些知识我们又能多么确定呢？软件系统应该遵循我们对物理世界所期望的法则，如因果关系吗？

幸运的是，我们不需要去搞清楚生命的意义。在分布式系统中，我们可以陈述关于行为（系统模型）的假设，并以满足这些假设的方式设计实际系统。算法可以被证明在某个系统模型中正确运行。这意味着即使底层系统模型提供了很少的保证，也可以实现可靠的行为。

但是，尽管可以使软件在不可靠的系统模型中表现良好，但这并不是可以直截了当实现的。在本章的其余部分中，我们将进一步探讨分布式系统中的知识和真理的概念，这将有助于我们思考我们可以做出的各种假设以及我们可能希望提供的保证。在第9章中，我们将着眼于分布式系统的一些例子，这些算法在特定的假设条件下提供了特定的保证。

真理由多数所定义

设想一个具有不对称故障的网络：一个节点能够接收发送给它的所有消息，但是来自该节点的任何传出消息被丢弃或延迟【19】。即使该节点运行良好，并且正在接收来自其他节点的请求，其他节点也无法听到其响应。经过一段时间后，其他节点宣布它已经死亡，因为他们没有听到节点的消息。这种情况就像梦魇一样：半断开（**semi-disconnected**）的节点被拖向墓地，敲打尖叫道“我没死！”——但是由于没有人能听到它的尖叫，葬礼队伍继续以坚忍的决心继续行进。

在一个稍微不那么梦魇的场景中，半断开的节点可能会注意到它发送的消息没有被其他节点确认，因此意识到网络中必定存在故障。尽管如此，节点被其他节点错误地宣告为死亡，而半连接的节点对此无能为力。

第三种情况，想象一个经历了一个长时间停止世界垃圾收集暂停（**stop-the-world GC Pause**）的节点。节点的所有线程被GC抢占并暂停一分钟，因此没有请求被处理，也没有响应被发送。其他节点等待，重试，不耐烦，并最终宣布节点死亡，并将其丢到灵车上。最后，GC完成，节点的线程继续，好像什么也没有发生。其他节点感到惊讶，因为所谓的死亡节点突然从棺材中抬起头来，身体健康，开始和旁观者高兴地聊天。GC后的节点最初甚至没有意识到已经经过了整整一分钟，而且自己已被宣告死亡。从它自己的角度来看，从最后一次与其他节点交谈以来，几乎没有经过任何时间。

这些故事的寓意是，节点不一定能相信自己对于情况的判断。分布式系统不能完全依赖单个节点，因为节点可能随时失效，可能会使系统卡死，无法恢复。相反，许多分布式算法都依赖于法定人数，即在节点之间进行投票（参阅“读写法定人数”）：决策需要来自多个节点的最小投票数，以减少对于某个特定节点的依赖。

这也包括关于宣告节点死亡的决定。如果法定数量的节点宣告另一个节点已经死亡，那么即使该节点仍感觉自己活着，它也必须被认为是死的。个体节点必须遵守法定决定并下台。

最常见的法定人数是超过一半的绝对多数（尽管其他类型的法定人数也是可能的）。多数法定人数允许系统继续工作，如果单个节点发生故障（三个节点可以容忍单节点故障；五个节点可以容忍双节点故障）。系统仍然是安全的，因为在这个制度中只能有一个多数——不能同时存在两个相互冲突的多数决定。当我们在第9章中讨论共识算法（**consensus algorithms**）时，我们将更详细地讨论法定人数的应用。

领导者与锁定

通常情况下，一些东西在一个系统中只能有一个。例如：

- 数据库分区的领导者只能有一个节点，以避免脑裂（**split brain**）（参阅“处理节点宕机”）。
- 特定资源的锁或对象只允许一个事务/客户端持有，以防同时写入和损坏。
- 一个特定的用户名只能被一个用户所注册，因为用户名必须唯一标识一个用户。

在分布式系统中实现这一点需要注意：即使一个节点认为它是“天选者（**the choosen one**）”（分区的负责人，锁的持有者，成功获取用户名的用户的请求处理程序），但这并不一定意味着有法定人数的节点同意！一个节点可能以前是领导者，但是如果其他节点在此期间宣布它死亡（例如，由于网络中断或GC暂停），则它可能已被降级，且另一个领导者可能已经当选。

如果一个节点继续表现为天选者，即使大多数节点已经声明它已经死了，则在考虑不周的系统中可能会导致问题。这样的节点能以自己赋予的权能向其他节点发送消息，如果其他节点相信，整个系统可能会做一些不正确的事情。

例如，图8-4显示了由于不正确的锁实现导致的数据损坏错误。（这个错误不仅仅是理论上的：HBase曾经有这个问题【74,75】）假设你要确保一个存储服务中的文件一次只能被一个客户访问，因为如果多个客户试图写对此，该文件将被损坏。您尝试通过在访问文件之前要求客户端从锁定服务获取租约来实现此目的。

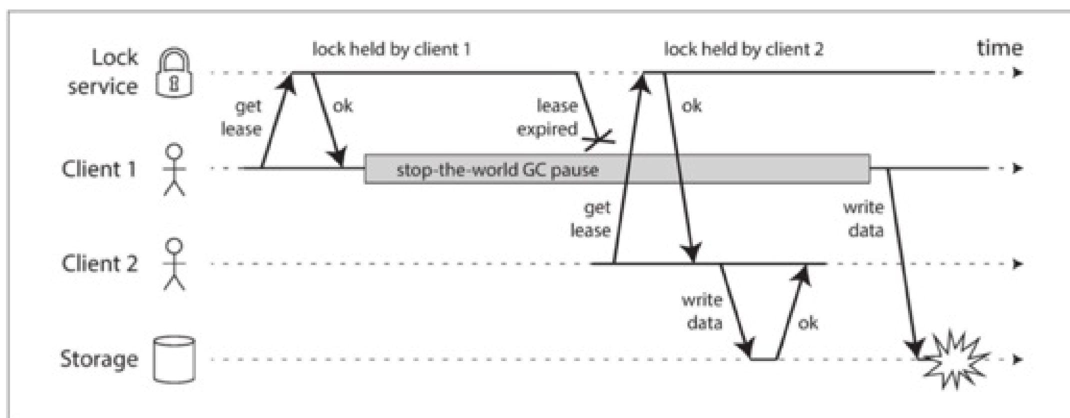


图**8-4** 分布式锁的实现不正确：客户端**1**认为它仍然具有有效的租约，即使它已经过期，从而破坏了存储中的文件

这个问题就是我们先前在“进程暂停”中讨论过的一个例子：如果持有租约的客户端暂停太久，它的租约将到期。另一个客户端可以获得同一文件的租约，并开始写入文件。当暂停的客户端回来时，它认为（不正确）它仍然有一个有效的租约，并继续写入文件。结果，客户的写入冲突和损坏的文件。

防护令牌

当使用锁或租约来保护对某些资源（如图8-4中的文件存储）的访问时，需要确保一个被误认为自己是“天选者”的节点不能中断系统的其它部分。实现这一目标的一个相当简单的技术就是防护（**fencing**），如图8-5所示

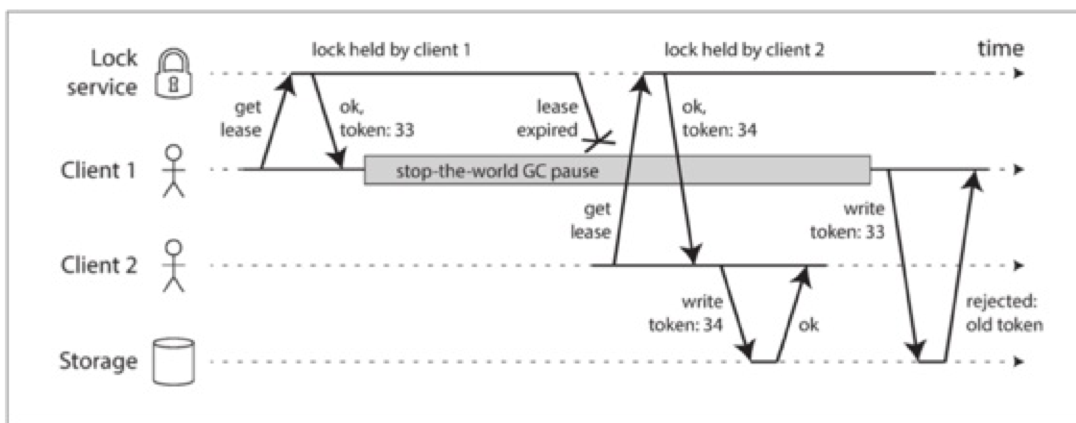


图**8-5** 只允许以增加屏蔽令牌的顺序进行写操作，从而保证存储安全

我们假设每次锁定服务器授予锁或租约时，它还会返回一个防护令牌（**fencing token**），这个数字在每次授予锁定时都会增加（例如，由锁定服务增加）。然后，我们可以要求客户端每次向存储服务发送写入请求时，都必须包含当前的屏蔽令牌。

在图8-5中，客户端1以33的令牌获得租约，但随后进入一个长时间的停顿并且租约到期。客户端2以34的令牌（该数字总是增加）获取租约，然后将其写入请求发送到存储服务，包括34的令牌。稍后，客户端1恢复生机并将其写入存储服务，包括其令牌值33.但是，存储服务器会记住它已经处理了一个具有更高令牌编号（34）的写入，因此它会拒绝带有令牌33的请求。

如果将ZooKeeper用作锁定服务，则可将事务标识 `zxid` 或节点版本 `cversion` 用作屏蔽令牌。由于它们保证单调递增，因此它们具有所需的属性【74】。

请注意，这种机制要求资源本身在检查令牌方面发挥积极作用，通过拒绝使用旧的令牌，而不是已经被处理的令牌来进行写操作——仅仅依靠客户端检查自己的锁状态是不够的。对于不明确支持屏蔽令牌的资源，可能仍然可以解决此限制（例如，在文件存储服务的情况下，可以将防护令牌包含在文件名中）。但是，为了避免在锁的保护之外处理请求，需要进行某种检查。

在服务器端检查一个令牌可能看起来像是一个缺点，但这可以说是一件好事：一个服务假定它的客户总是守规矩并不明智，因为使用客户端的人与运行服务的人优先级非常不一样【76】。因此，任何服务保护自己免受意外客户的滥用是一个好主意。

拜占庭故障

屏蔽令牌可以检测和阻止无意中发生错误的节点（例如，因为它尚未发现其租约已过期）。但是，如果节点有意破坏系统的保证，则可以通过使用假屏蔽令牌发送消息来轻松完成此操作。

在本书中，我们假设节点是不可靠但诚实的：它们可能很慢或者从不响应（由于故障），并且它们的状态可能已经过时（由于GC暂停或网络延迟），但是我们假设如果节点它做出了回应，它正在说出“真相”：尽其所知，它正在按照协议的规则扮演其角色。

如果存在节点可能“撒谎”（发送任意错误或损坏的响应）的风险，则分布式系统的问题变得更困难了——例如，如果节点可能声称其实际上没有收到特定的消息。这种行为被称为拜占庭故障（**Byzantine fault**），在不信任的环境中达成共识的问题被称为拜占庭将军问题【77】。

> ### 拜占庭将军问题
>
> 拜占庭将军问题是所谓“两将军问题”的概括【78】，它想象两个将军需要就战斗计划达成一致的情况。由于他们在两个不同的地点建立了营地，他们只能通过信使进行沟通，信使有时会被延迟或丢失（就像网络中的信息包一样）。我们将在第9章讨论这个共识问题。
>
> 在这个拜占庭式的问题中，有n位将军需要同意，他们的努力因为有一些叛徒在他们中间而受到阻碍。大多数的将军都是忠诚的，因而发出了真实的信息，但是叛徒可能会试图通过发送虚假或不真实的信息来欺骗和混淆他人（在试图保持未被发现的同时）。事先并不知道叛徒是谁。
>
> 拜占庭是后来成为君士坦丁堡的古希腊城市，现在在土耳其的伊斯坦布尔。没有任何历史证据表明拜占庭将军比其他地方更容易出现阴谋和阴谋。相反，这个名字来源于拜占庭式的过度复杂，官僚，迂回等意义，早在计算机之前就已经在政治中被使用了【79】。Lamport想要选一个不会冒犯任何读者的国家，他被告知将其称为阿尔巴尼亚将军问题并不是一个好主意【80】。

当一个系统在部分节点发生故障、不遵守协议、甚至恶意攻击、扰乱网络时仍然能继续正确工作，称之为拜占庭容错（**Byzantine fault-tolerant**）的，在特定场景下，这种担忧在是有意义的：

- 在航空航天环境中，计算机内存或CPU寄存器中的数据可能被辐射破坏，导致其以任意不可预知的方式响应其他节点。由于系统故障将非常昂贵（例如，飞机撞毁和炸死船上所有人员，或火箭与国际空间站相撞），飞行控制系统必须容忍拜占庭故障【81,82】。
- 在多个参与组织的系统中，一些参与者可能会试图欺骗或欺骗他人。在这种情况下，节点仅仅信任另一个节点的消息是不安全的，因为它们可能是出于恶意的目的而被发送的。例如，像比特币和其他区块链一样的对等网络可以被认为是让互不信任的各方同意交易是否发生的一种方式，而不依赖于中央当局【83】。

然而，在本书讨论的那些系统中，我们通常可以安全地假设没有拜占庭式的错误。在你的数据中心里，所有的节点都是由你的组织控制的（所以他们可以信任），辐射水平足够低，内存损坏不是一个大问题。制作拜占庭容错系统的协议相当复杂【84】，而容错嵌入式系统依赖于硬件层面的支持【81】。在大多数服务器端数据系统中，部署拜占庭容错解决方案的成本使其变得不切实际。

Web应用程序确实需要预期受终端用户控制的客户端（如Web浏览器）的任意和恶意行为。这就是为什么输入验证，清理和输出转义如此重要：例如，防止SQL注入和跨站点脚本。但是，我们通常不使用拜占庭容错协议，而只是让服务器决定什么是客户端行为，而不是允许的。在没有这种中心授权的对等网络中，拜占庭容错更为重要。

软件中的一个错误可能被认为是拜占庭式的错误，但是如果您将相同的软件部署到所有节点上，那么拜占庭式的容错算法不能为您节省。大多数拜占庭式容错算法要求超过三分之二的节点能够正常工作（即，如果有四个节点，最多只能有一个故障）。要使用这种方法对付bug，你必须有四个独立的相同软件的实现，并希望一个bug只出现在四个实现之一中。

同样，如果一个协议可以保护我们免受漏洞，安全妥协和恶意攻击，那么这将是有吸引力的。不幸的是，这也是不现实的：在大多数系统中，如果攻击者可以渗透一个节点，那他们可能会渗透所有这些节点，因为它们可能运行相同的软件。因此传统机制（认证，访问控制，加密，防火墙等）仍然是攻击者的主要保护措施。

弱谎言形式

尽管我们假设节点通常是诚实的，但值得向软件中添加防止“撒谎”弱形式的机制——例如，由硬件问题导致的无效消息，软件错误和错误配置。这种保护机制并不是完全的拜占庭容错，因为它们不能抵挡决心坚定的对手，但它们仍然是简单而实用的步骤，以提高可靠性。例如：

- 由于硬件问题或操作系统，驱动程序，路由器等中的错误，网络数据包有时会受到损坏。通常，内建于TCP和UDP中的校验和会俘获损坏的数据包，但有时它们会逃避检测【85,86,87】 。简单的措施通常是采用充分的保护来防止这种破坏，例如应用程序级协议中的校验和。
- 可公开访问的应用程序必须仔细清理来自用户的任何输入，例如检查值是否在合理的范围内，并限制字符串的大小以防止通过大内存分配拒绝服务。防火墙后面的内部服务可能能够在对输入进行较不严格的检查的情况下逃脱，但是一些基本的理智检查（例如，在协议解析中）是一个好主意。
- NTP客户端可以配置多个服务器地址。同步时，客户端联系所有的服务器，估计它们的误差，并检查大多数服务器是否在对某个时间范围内达成一致。只要大多数的服务器没问题，一个配置错误的NTP服务器报告的时间会被当成特异值从同步中排除【37】。使用多个服务器使NTP更健壮（比起只用单个服务器来）。

系统模型与现实

已经有很多算法被设计以解决分布式系统问题——例如，我们将在第9章讨论共识问题的解决方案。为了有用，这些算法需要容忍我们在本章中讨论的分布式系统的各种故障。

算法的编写方式并不过分依赖于运行的硬件和软件配置的细节。这又要求我们以某种方式将我们期望在系统中发生的错误形式化。我们通过定义一个系统模型来做到这一点，这个模型是一个抽象，描述一个算法可能承担的事情。 关于定时假设，三种系统模型是常用的：

同步模型

同步模型（**synchronous model**）假设网络延迟，进程暂停和和时钟误差都是有界限的。这并不意味着完全同步的时钟或零网络延迟；这只意味着你知道网络延迟，暂停和时钟漂移将永远不会超过某个固定的上限【88】。同步模型并不是大多数实际系统的现实模型，因为（如本章所讨论的）无限延迟和暂停确实会发生。

部分同步模型

部分同步（**partial synchronous**）意味着一个系统在大多数情况下像一个同步系统一样运行，但有时候会超出网络延迟，进程暂停和时钟漂移的界限【88】。这是很多系统的现实模型：大多数情况下，网络和进程表现良好，否则我们永远无法完成任何事情，但是我们必须承认，在任何时刻假设都存在偶然被破坏的事实。发生这种情况时，网络延迟，暂停和时钟错误可能会变得相当大。

异步模型

在这个模型中，一个算法不允许对时机做任何假设——事实上它甚至没有时钟（所以它不能使用超时）。一些算法被设计为可用于异步模型，但非常受限。

进一步来说，除了时间问题，我们还要考虑节点失效。三种最常见的节点系统模型是：

崩溃-停止故障

在崩溃停止（**crash-stop**）模型中，算法可能会假设一个节点只能以一种方式失效，即通过崩溃。这意味着节点可能在任意时刻突然停止响应，此后该节点永远消失——它永远不会回来。

崩溃-恢复故障

我们假设节点可能会在任何时候崩溃，但也许会在未知的时间之后再次开始响应。在崩溃-恢复（**crash-recovery**）模型中，假设节点具有稳定的存储（即，非易失性磁盘存储）且会在崩溃中保留，而内存中的状态会丢失。

拜占庭（任意）故障

节点可以做（绝对意义上的）任何事情，包括试图戏弄和欺骗其他节点，如上一节所述。

对于真实系统的建模，具有崩溃-恢复故障（**crash-recovery**）的部分同步模型（**partial synchronous**）通常是最有用的模型。分布式算法如何应对这种模型？

算法的正确性

为了定义算法是正确的，我们可以描述它的属性。例如，排序算法的输出具有如下特性：对于输出列表中的任何两个不同的元素，左边的元素比右边的元素小。这只是定义对列表进行排序含义的一种形式方式。

同样，我们可以写下我们想要的分布式算法的属性来定义它的正确含义。例如，如果我们正在为一个锁生成屏蔽令牌（参阅“屏蔽令牌”），我们可能要求算法具有以下属性：

唯一性

没有两个屏蔽令牌请求返回相同的值。

单调序列

如果请求 x 返回了令牌 t_x，并且请求y返回了令牌t_y，并且 x 在 y 开始之前已经完成，那么$t_x <t_y$。

可用性

请求防护令牌并且不会崩溃的节点，最终会收到响应。

如果一个系统模型中的算法总是满足它在我们假设可能发生的所有情况下的性质，那么这个算法是正确的。但这如何有意义？如果所有的节点崩溃，或者所有的网络延迟突然变得无限长，那么没有任何算法能够完成任何事情。

安全性和活性

为了澄清这种情况，有必要区分两种不同的性质：安全性（**safety**）和活性（**liveness**）。在刚刚给出的例子中，唯一性（**uniqueness**）和单调序列（**monotonic sequence**）是安全属性，但可用性是活性（**liveness**）属性。

这两种性质有什么区别？一个试金石就是，活性属性通常在定义中通常包括“最终”一词。（是的，你猜对了——最终一致性是一个活性属性【89】。）

安全性通常被非正式地定义为，没有坏事发生，而活性通常就类似：最终好事发生。但是，最好不要过多地阅读那些非正式的定义，因为好与坏的含义是主观的。安全性和活性的实际定义是精确的和数学的【90】：

- 如果安全属性被违反，我们可以指向一个特定的时间点（例如，如果违反了唯一性属性，我们可以确定重复的防护令牌返回的特定操作）。违反安全属性后，违规行为不能撤销——损失已经发生。
- 活性属性反过来：在某个时间点（例如，一个节点可能发送了一个请求，但还没有收到响应），它可能不成立，但总是希望在未来（即通过接受答复）。

区分安全性和活性属性的一个优点是可以帮助我们处理困难的系统模型。对于分布式算法，在系统模型的所有可能情况下，要求始终保持安全属性是常见的【88】。也就是说，即使所有节点崩溃，或者整个网络出现故障，算法仍然必须确保它不会返回错误的结果（即保证安全性得到满足）。

但是，对于活性属性，我们可以提出一些注意事项：例如，只有在大多数节点没有崩溃的情况下，只有当网络最终从中断中恢复时，我们才可以说请求需要接收响应。部分同步模型的定义要求系统最终返回到同步状态——即任何网络中断的时间段只会持续一段有限的时间，

然后进行修复。

将系统模型映射到现实世界

安全性和活性属性以及系统模型对于推理分布式算法的正确性非常有用。然而，在实践中实施算法时，现实的混乱事实再一次地让你咬牙切齿，很明显系统模型是对现实的简化抽象。

例如，在故障恢复模型中的算法通常假设稳定存储器中的数据经历了崩溃。但是，如果磁盘上的数据被破坏，或者由于硬件错误或错误配置导致数据被清除，会发生什么情况？如果服务器存在固件错误并且在重新启动时无法识别其硬盘驱动器，即使驱动器已正确连接到服务器，也会发生什么情况？

法定人数算法（参见“读写法定人数”）依赖节点来记住它声称存储的数据。如果一个节点可能患有健忘症，忘记了以前存储的数据，这会打破法定条件，从而破坏算法的正确性。也许需要一个新的系统模型，在这个模型中，我们假设稳定的存储大多存在崩溃，但有时可能会丢失。但是那个模型就变得更难以推理了。

算法的理论描述可以简单宣称一些事在假设上是不会发生的——在非拜占庭式系统中。但实际上我们还是需要对可能发生和不可能发生的故障做出假设，真实世界的实现，仍然会包括处理“假设上不可能”情况的代码，即使代码可能就是 `printf("you sucks")` 和 `exit(666)` ，实际上也就是留给运维来擦屁股。（这可以说是计算机科学和软件工程间的一个差异）。

这并不是说理论上抽象的系统模型是毫无价值的，恰恰相反。它们对于将实际系统的复杂性降低到一个我们可以推理的可处理的错误是非常有帮助的，以便我们能够理解这个问题，并试图系统地解决这个问题。我们可以证明算法是正确的，通过显示它们的属性总是保持在某个系统模型中

证明算法正确并不意味着它在真实系统上的实现必然总是正确的。但这迈出了很好的第一步，因为理论分析可以发现算法中的问题，这种问题可能会在现实系统中长期潜伏，直到你的假设（例如，时间）因为不寻常的情况被打破。理论分析与经验测试同样重要。

本章小结

在本章中，我们讨论了分布式系统中可能发生的各种问题，包括：

- 当您尝试通过网络发送数据包时，数据包可能会丢失或任意延迟。同样，答复可能会丢失或延迟，所以如果你没有得到答复，你不知道消息是否通过。
- 节点的时钟可能会与其他节点显着不同步（尽管您尽最大努力设置NTP），它可能会突然跳转或跳回，依靠它是很危险的，因为您很可能没有好的测量你的时钟的错误间隔。
- 一个进程可能会在其执行的任何时候暂停一段相当长的时间（可能是因为世界上的垃圾收集器），被其他节点宣告死亡，然后再次复活，却没有意识到它被暂停了。

这类部分失效可能发生的事实是分布式系统的决定性特征。每当软件试图做任何涉及其他节点的事情时，偶尔就有可能会失败，或者随机变慢，或者根本没有响应（最终超时）。在分布式系统中，我们试图在软件中建立部分失效的容错机制，这样整个系统即使在某些组成部分被破坏的情况下，也可以继续运行。

为了容忍错误，第一步是检测它们，但即使这样也很难。大多数系统没有检测节点是否发生故障的准确机制，所以大多数分布式算法依靠超时来确定远程节点是否仍然可用。但是，超时无法区分网络失效和节点失效，并且可变的网络延迟有时会导致节点被错误地怀疑发生故障。此外，有时一个节点可能处于降级状态：例如，由于驱动程序错误【94】，千兆网卡可能突然下降到1 Kb/s的吞吐量。这样一个“跛行”而不是死掉的节点可能比一个干净的失效节点更难处理。

一旦检测到故障，使系统容忍它也并不容易：没有全局变量，没有共享内存，没有共同的知识，或机器之间任何其他种类的共享状态。节点甚至不能就现在是什么时间达成一致，就不用说更深奥的了。信息从一个节点流向另一个节点的唯一方法是通过不可靠的网络发送信息。重大决策不能由一个节点安全地完成，因此我们需要一个能从其他节点获得帮助的协议，并争取达到法定人数以达成一致。

如果你习惯于在理想化的数学完美（同一个操作总能确定地返回相同的结果）的单机环境中编写软件，那么转向分布式系统的凌乱的物理现实可能会有些令人震惊。相反，如果能够在单台计算机上解决一个问题，那么分布式系统工程师通常会认为这个问题是平凡的【5】，现在单个计算机确实可以做很多事情【95】。如果你可以避免打开潘多拉的盒子，把东西放在一台机器上，那么通常是值得的。

但是，正如在第二部分的介绍中所讨论的那样，可扩展性并不是使用分布式系统的唯一原因。容错和低延迟（通过将数据放置在距离用户较近的地方）是同等重要的目标，而这些不能用单个节点实现。

在本章中，我们也转换了几次话题，探讨了网络，时钟和进程的不可靠性是否是不可避免的自然规律。我们看到这并不是：有可能给网络提供硬实时的响应保证和有限的延迟，但是这样做非常昂贵，且导致硬件资源的利用率降低。大多数非安全关键系统会选择便宜而不可靠，而不是昂贵和可靠。

我们还谈到了超级计算机，它们采用可靠的组件，因此当组件发生故障时必须完全停止并重新启动。相比之下，分布式系统可以永久运行而不会在服务层面中断，因为所有的错误和维护都可以在节点级别进行处理——至少在理论上是如此。（实际上，如果一个错误的配置变更被应用到所有的节点，仍然会使分布式系统瘫痪）。

本章一直在讲存在的问题，给我们展现了一幅黯淡的前景。在下一章中，我们将继续讨论解决方案，并讨论一些旨在解决分布式系统中所有问题的算法。

参考文献

1. Mark Cavage: Just No Getting Around It: You’re Building a Distributed System] (http://queue.acm.org/detail.cfm?id=2482856),” *ACM Queue*, volume 11, number 4, pages 80-89, April 2013. doi:10.1145/2466486.2482856
2. Jay Kreps: “Getting Real About Distributed System Reliability,” *blog.empathybox.com*, March 19, 2012.
3. Sydney Padua: *The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer*. Particular Books, April ISBN: 978-0-141-98151-2
4. Coda Hale: “You Can’t Sacrifice Partition Tolerance,” *codahale.com*, October 7, 2010.
5. Jeff Hodges: “Notes on Distributed Systems for Young Bloods,” *somethingsimilar.com*, January 14, 2013.
6. Antonio Regalado: “Who Coined 'Cloud Computing’?,” *technologyreview.com*, October 31, 2011.
7. Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle: “The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition,” *Synthesis Lectures on Computer Architecture*, volume 8, number 3, Morgan & Claypool Publishers, July 2013.doi:10.2200/S00516ED2V01Y201306CAC024, ISBN: 978-1-627-05010-4
8. David Fiala, Frank Mueller, Christian Engelmann, et al.: “Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing,” at *International Conference for High Performance Computing, Networking, Storage and Analysis* (SC12), November 2012.
9. Arjun Singh, Joon Ong, Amit Agarwal, et al.: “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network,” at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), August 2015. doi:10.1145/2785956.2787508
10. Glenn K. Lockwood: “Hadoop's Uncomfortable Fit in HPC,” *glennklockwood.blogspot.co.uk*, May 16, 2014.
11. John von Neumann: “Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components,” in *Automata Studies (AM-34)*, edited by Claude E. Shannon and John McCarthy, Princeton University Press, 1956. ISBN: 978-0-691-07916-5
12. Richard W. Hamming: *The Art of Doing Science and Engineering*. Taylor & Francis, 1997. ISBN: 978-9-056-99500-3
13. Claude E. Shannon: “A Mathematical Theory of Communication,” *The Bell System Technical Journal*, volume 27, number 3, pages 379–423 and 623–656, July 1948.
14. Peter Bailis and Kyle Kingsbury: “The Network Is Reliable,” *ACM Queue*, volume 12, number 7, pages 48-55, July 2014. doi:10.1145/2639988.2639988
15. Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish: “Taming Uncertainty in Distributed Systems with Help from the Network,” at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741976

16. Phillipa Gill, Navendu Jain, and Nachiappan Nagappan: "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," at *ACM SIGCOMM Conference*, August 2011. doi:10.1145/2018436.2018477
17. Mark Imbriaco: "Downtime Last Saturday," *github.com*, December 26, 2012.
18. Will Oremus: "The Global Internet Is Being Attacked by Sharks, Google Confirms," *slate.com*, August 15, 2014.
19. Marc A. Donges: "Re: bnx2 cards Intermittantly Going Offline," Message to Linux *netdev* mailing list, *spinics.net*, September 13, 2012.
20. Kyle Kingsbury: "Call Me Maybe: Elasticsearch," *aphyr.com*, June 15, 2014.
21. Salvatore Sanfilippo: "A Few Arguments About Redis Sentinel Properties and Fail Scenarios," *antirez.com*, October 21, 2014.
22. Bert Hubert: "The Ultimate SO_LINGER Page, or: Why Is My TCP Not Reliable," *blog.netherlabs.nl*, January 18, 2009.
23. Nicolas Liochon: "CAP: If All You Have Is a Timeout, Everything Looks Like a Partition," *blog.thislongrun.com*, May 25, 2015.
24. Jerome H. Saltzer, David P. Reed, and David D. Clark: "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984. doi:10.1145/357401.357402
25. Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, et al.: "Queues Don't Matter When You Can JUMP Them!," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.
26. Guohui Wang and T. S. Eugene Ng: "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," at *29th IEEE International Conference on Computer Communications* (INFOCOM), March 2010. doi:10.1109/INFCOM.2010.5461931
27. Van Jacobson: "Congestion Avoidance and Control," at *ACM Symposium on Communications Architectures and Protocols* (SIGCOMM), August 1988. doi:10.1145/52324.52356
28. Brandon Philips: "etcd: Distributed Locking and Service Discovery," at *Strange Loop*, September 2014.
29. Steve Newman: "A Systematic Look at EC2 I/O," *blog.scalyr.com*, October 16, 2012.
30. Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama: "The ϕ Accrual Failure Detector," Japan Advanced Institute of Science and Technology, School of Information Science, Technical Report IS-RR-2004-010, May 2004.
31. Jeffrey Wang: "Phi Accrual Failure Detector," *ternarysearch.blogspot.co.uk*, August 11, 2013.
32. Srinivasan Keshav: *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley Professional, May 1997. ISBN: 978-0-201-63442-6
33. Cisco, "Integrated Services Digital Network," *docwiki.cisco.com*.

34. Othmar Kyas: *ATM Networks*. International Thomson Publishing, 1995. ISBN: 978-1-850-32128-6
35. "InfiniBand FAQ," Mellanox Technologies, December 22, 2014.
36. Jose Renato Santos, Yoshio Turner, and G. (John) Janakiraman: "End-to-End Congestion Control for InfiniBand," at *22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM), April 2003. Also published by HP Laboratories Palo Alto, Tech Report HPL-2002-359. doi:10.1109/INFCOM.2003.1208949
37. Ulrich Windl, David Dalton, Marc Martinec, and Dale R. Worley: "The NTP FAQ and HOWTO," *ntp.org*, November 2006.
38. John Graham-Cumming: "How and why the leap second affected Cloudflare DNS," *blog.cloudflare.com*, January 1, 2017.
39. David Holmes: "Inside the Hotspot VM: Clocks, Timers and Scheduling Events – Part I – Windows," *blogs.oracle.com*, October 2, 2006.
40. Steve Loughran: "Time on Multi-Core, Multi-Socket Servers," *steveloughran.blogspot.co.uk*, September 17, 2015.
41. James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: "Spanner: Google's Globally-Distributed Database," at *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.
42. M. Caporaloni and R. Ambrosini: "How Closely Can a Personal Computer Clock Track the UTC Timescale Via the Internet?," *European Journal of Physics*, volume 23, number 4, pages L17–L21, June 2012. doi:10.1088/0143-0807/23/4/103
43. Nelson Minar: "A Survey of the NTP Network," *alumni.media.mit.edu*, December 1999.
44. Viliam Holub: "Synchronizing Clocks in a Cassandra Cluster Pt. 1 – The Problem," *blog.logentries.com*, March 14, 2014.
45. Poul-Henning Kamp: "The One-Second War (What Time Will You Die?)," *ACM Queue*, volume 9, number 4, pages 44–48, April 2011. doi:10.1145/1966989.1967009
46. Nelson Minar: "Leap Second Crashes Half the Internet," *somebits.com*, July 3, 2012.
47. Christopher Pascoe: "Time, Technology and Leaping Seconds," *googleblog.blogspot.co.uk*, September 15, 2011.
48. Mingxue Zhao and Jeff Barr: "Look Before You Leap – The Coming Leap Second and AWS," *aws.amazon.com*, May 18, 2015.
49. Darryl Veitch and Kanthaiah Vijayalayan: "Network Timing and the 2015 Leap Second," at *17th International Conference on Passive and Active Measurement* (PAM), April 2016. doi:10.1007/978-3-319-30505-9_29
50. "Timekeeping in VMware Virtual Machines," Information Guide, VMware, Inc., December 2011.
51. "MiFID II / MiFIR: Regulatory Technical and Implementing Standards – Annex I (Draft)," European Securities and Markets Authority, Report ESMA/2015/1464, September 2015.
52. Luke Bigum: "Solving MiFID II Clock Synchronisation With Minimum Spend (Part 1),"

lmax.com, November 27, 2015.

53. Kyle Kingsbury: “Call Me Maybe: Cassandra,” *aphyr.com*, September 24, 2013.
54. John Daily: “Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems,” *basho.com*, November 12, 2013.
55. Kyle Kingsbury: “The Trouble with Timestamps,” *aphyr.com*, October 12, 2013.
56. Leslie Lamport: “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563
57. Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa, et al.: “Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases,” State University of New York at Buffalo, Computer Science and Engineering Technical Report 2014-04, May 2014.
58. Justin Sheehy: “There Is No Now: Problems With Simultaneity in Distributed Systems,” *ACM Queue*, volume 13, number 3, pages 36–41, March 2015. doi:10.1145/2733108
59. Murat Demirbas: “Spanner: Google's Globally-Distributed Database,” *muratbuffalo.blogspot.co.uk*, July 4, 2013.
60. Dahlia Malkhi and Jean-Philippe Martin: “Spanner's Concurrency Control,” *ACM SIGACT News*, volume 44, number 3, pages 73–77, September 2013. doi:10.1145/2527748.2527767
61. Manuel Bravo, Nuno Diegues, Jingna Zeng, et al.: “On the Use of Clocks to Enforce Consistency in the Cloud,” *IEEE Data Engineering Bulletin*, volume 38, number 1, pages 18–31, March 2015.
62. Spencer Kimball: “Living Without Atomic Clocks,” *cockroachlabs.com*, February 17, 2016.
63. Cary G. Gray and David R. Cheriton:“Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency,” at *12th ACM Symposium on Operating Systems Principles* (SOSP), December 1989. doi:10.1145/74850.74870
64. Todd Lipcon: “Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1,” *blog.cloudera.com*, February 24, 2011.
65. Martin Thompson: “Java Garbage Collection Distilled,” *mechanical-sympathy.blogspot.co.uk*, July 16, 2013.
66. Alexey Ragozin: “How to Tame Java GC Pauses? Surviving 16GiB Heap and Greater,” *java.dzone.com*, June 28, 2011.
67. Christopher Clark, Keir Fraser, Steven Hand, et al.: “Live Migration of Virtual Machines,” at *2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation* (NSDI), May 2005.
68. Mike Shaver: “fsyncers and Curveballs,” *shaver.off.net*, May 25, 2008.
69. Zhenyun Zhuang and Cuong Tran: “Eliminating Large JVM GC Pauses Caused by Background IO Traffic,” *engineering.linkedin.com*, February 10, 2016.
70. David Terei and Amit Levy: “Blade: A Data Center Garbage Collector,”

arXiv:1504.02578, April 13, 2015.

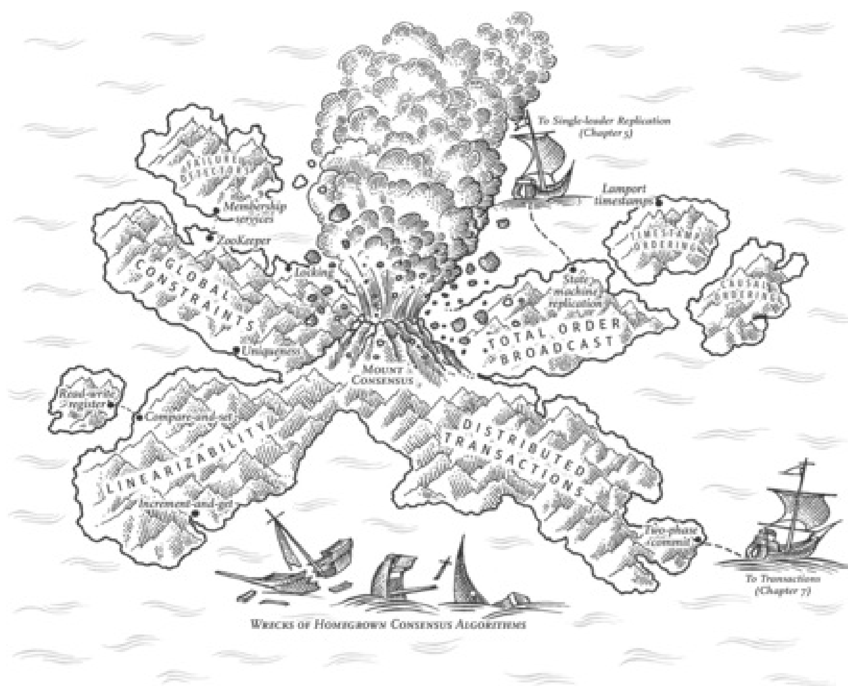
71. Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz: "Trash Day: Coordinating Garbage Collection in Distributed Systems," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.
72. "Predictable Low Latency," Cinnober Financial Technology AB, *cinnober.com*, November 24, 2013.
73. Martin Fowler: "The LMAX Architecture," *martinfowler.com*, July 12, 2011.
74. Flavio P. Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN: 978-1-449-36130-3
75. Enis Söztutar: "HBase and HDFS: Understanding Filesystem Usage in HBase," at *HBaseCon*, June 2013.
76. Caitie McCaffrey: "Clients Are Jerks: AKA How Halo 4 DoSed the Services at Launch & How We Survived," *caitiem.com*, June 23, 2015.
77. Leslie Lamport, Robert Shostak, and Marshall Pease: "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 4, number 3, pages 382–401, July 1982. doi:10.1145/357172.357176
78. Jim N. Gray: "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, volume 60, edited by R. Bayer, R. M. Graham, and G. Seegmüller, pages 393–481, Springer-Verlag, 1978. ISBN: 978-3-540-08755-7
79. Brian Palmer: "How Complicated Was the Byzantine Empire?," *slate.com*, October 20, 2011.
80. Leslie Lamport: "My Writings," *research.microsoft.com*, December 16, 2014. This page can be found by searching the web for the 23-character string obtained by removing the hyphens from the string `allla-mport-spubso-ntheweb` .
81. John Rushby: "Bus Architectures for Safety-Critical Embedded Systems," at *1st International Workshop on Embedded Software* (EMSOFT), October 2001.
82. Jake Edge: "ELC: SpaceX Lessons Learned," *lwn.net*, March 6, 2013.
83. Andrew Miller and Joseph J. LaViola, Jr.: "Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin," University of Central Florida, Technical Report CS-TR-14-01, April 2014.
84. James Mickens: "The Saddest Moment," *USENIX ;login: logout*, May 2013.
85. Evan Gilman: "The Discovery of Apache ZooKeeper's Poison Packet," *pagerduty.com*, May 7, 2015.
86. Jonathan Stone and Craig Partridge: "When the CRC and TCP Checksum Disagree," at *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM), August 2000. doi:10.1145/347059.347561
87. Evan Jones: "How Both TCP and Ethernet Checksums Fail," *evanjones.ca*, October 5, 2015.
88. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of

Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

89. Peter Bailis and Ali Ghodsi: "Eventual Consistency Today: Limitations, Extensions, and Beyond," *ACM Queue*, volume 11, number 3, pages 55-63, March 2013. doi:10.1145/2460276.2462076
90. Bowen Alpern and Fred B. Schneider: "Defining Liveness," *Information Processing Letters*, volume 21, number 4, pages 181–185, October 1985. doi:10.1016/0020-0190(85)90056-090056-0)
91. Flavio P. Junqueira: "Dude, Where's My Metadata?," *fpj.me*, May 28, 2015.
92. Scott Sanders: "January 28th Incident Report," *github.com*, February 3, 2016.
93. Jay Kreps: "A Few Notes on Kafka and Jepsen," *blog.empathybox.com*, September 25, 2013.
94. Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: "Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems," at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523627
95. Frank McSherry, Michael Isard, and Derek G. Murray: "Scalability! But at What COST?," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

上一章	目录	下一章
第七章：事务	设计数据密集型应用	第九章：一致性与共识

9. 一致性与共识



> 好死不如赖活着 —— Jay Kreps, 关于Kafka与 Jepsen的若干笔记 (2013)

[TOC]

正如第8章所讨论的，分布式系统中的许多事情可能会出错。处理这种故障的最简单方法是简单地让整个服务失效，并向用户显示错误消息。如果无法接受这个解决方案，我们就需要找到容错的方法—— 即使某些内部组件出现故障，服务也能正常运行。

在本章中，我们将讨论构建容错分布式系统的算法和协议的一些例子。我们将假设第8章的所有问题都可能发生：网络中的数据包可能会丢失，重新排序，重复递送或任意延迟；时钟只是尽其所能地近似；且节点可以暂停（例如，由于垃圾收集）或随时崩溃。

构建容错系统的最好方法，是找到一些带有实用保证的通用抽象，实现一次，然后让应用依赖这些保证。这与第7章中的事务处理方法相同：通过使用事务，应用可以假装没有崩溃（原子性），没有其他人同时访问数据库（隔离），存储设备是完全可靠的（持久性）。即使发生崩溃，竞态条件和磁盘故障，事务抽象隐藏了这些问题，因此应用程序不必担心它们。

现在我们将继续沿着同样的路线前进，寻求可以让应用忽略分布式系统部分问题的抽象概念。例如，分布式系统最重要的抽象之一就是共识（**consensus**）：就是让所有的节点对某件事达成一致。正如我们在本章中将会看到的那样，尽管存在网络故障和流程故障，可靠地达成共识是一个令人惊讶的棘手问题。

一旦达成共识，应用程序可以将其用于各种目的。例如，假设你有一个单主复制的数据库。如果主库挂点，并且需要故障转移到另一个节点，剩余的数据库节点可以使用共识来选举新的领导者。正如在“处理节点宕机”中所讨论的那样，重要的是只有一个领导者，且所有的节点都认同其领导。如果两个节点都认为自己是领导者，这种情况被称为脑裂（**split brain**），且经常导致数据丢失。正确实现共识有助于避免这种问题。

在本章后面的“分布式事务和共识”中，我们将研究解决共识和相关问题的算法。但首先，我们首先需要探索可以在分布式系统中提供的保证和抽象的范围。

我们需要了解可以做什么和不可以做什么的范围：在某些情况下，系统可以容忍故障并继续工作；在其他情况下，这是不可能的。我们将深入研究什么可能而什么不可能的限制，既通过理论证明，也通过实际实现。我们将在本章中概述这些基本限制。

分布式系统领域的研究人员几十年来一直在研究这些主题，所以有很多资料—— 我们只能介绍一些皮毛。在本书中，我们没有空间去详细介绍形式模型和证明的细节，所以我们将坚持非正式的直觉。如果你有兴趣，参考文献可以提供更多的深度。

一致性保证

在“复制延迟问题”中，我们看到了数据库复制中发生的一些时序问题。如果你在同一时刻查看两个数据库节点，则可能在两个节点上看到不同的数据，因为写请求在不同的时间到达不同的节点。无论数据库使用何种复制方法（单主复制，多主复制或无主复制），都会出现这些不一致情况。

大多数复制的数据库至少提供了最终一致性，这意味着如果你停止向数据库写入数据并等待一段不确定的时间，那么最终所有的读取请求都会返回相同的值【1】。换句话说，不一致性是暂时的，最终会自行解决（假设网络中的任何故障最终都会被修复）。最终一致性的一个更好的名字可能是收敛（**convergence**），因为我们预计所有的复本最终会收敛到相同的值【2】。

然而，这是一个非常弱的保证 —— 它并没有说什么什么时候副本会收敛。在收敛之前，读操作可能会返回任何东西或什么都没有【1】。例如，如果你写入了一个值，然后立即再次读取，这并不能保证你能看到刚跟写入的值，因为读请求可能会被路由到另外的副本上。（参阅“读己之写”）。

对于应用程序开发人员而言，最终一致性是很困难的，因为它与普通单线程程序中变量的行为有很大区别。如果将一个值赋给一个变量，然后很快地再次读取，你不会认为可能读到旧的值，或者读取失败。数据库表面上看起来像一个你可以读写的变量，但实际上它有更复杂的语义【3】。

在与只提供弱保证的数据库打交道时，你需要始终意识到它的局限性，而不是意外地作出太多假设。错误往往是微妙的，很难找到，也很难测试，因为应用可能在大多数情况下运行良好。当系统出现故障（例如网络中断）或高并发时，最终一致性的边缘情况才会显现出来。

本章将探索数据系统可能选择提供的更强一致性模型。它不是免费的：具有较强保证的系统可能会比保证较差的系统具有更差的性能或更少的容错性。尽管如此，更强的保证可以吸引人，因为它们更容易用对。只有见过不同的一致性模型后，才能更好地决定哪一个最适合自己的需求。

分布式一致性模型和我们之前讨论的事务隔离级别的层次结构有一些相似之处【4,5】（参见“弱隔离级别”）。尽管两者有一部分内容重叠，但它们大多是无关的问题：事务隔离主要是为了，避免由于同时执行事务而导致的竞争状态，而分布式一致性主要关于，面对延迟和故障时，如何协调副本间的状态。

本章涵盖了广泛的话题，但我们将会看到这些领域实际上是紧密联系在一起的：

- 首先看一下常用的最强一致性模型之一，线性一致性（**linearizability**），并考察其优缺点。
- 然后我们将检查分布式系统中事件顺序的问题，特别是因果关系和全局顺序的问题。
- 在第三部分（“分布式事务和共识”）中将探讨如何原子地提交分布式事务，这将最终引领我们走向共识问题的解决方案。

线性一致性

在最终一致的数据库，如果你在同一时刻问两个不同副本相同的问题，可能会得到两个不同的答案。这很让人困惑。如果数据库可以提供只有一个副本的假象（即，只有一个数据副本），那么事情就简单太多了。那么每个客户端都会有相同的数据视图，且不必担心复制滞后了。

这就是线性一致性（**linearizability**）背后的想法【6】（也称为原子一致性（**atomic consistency**）【7】，强一致性（**strong consistency**），立即一致性（**immediate consistency**）或外部一致性（**external consistency** ）【8】）。线性一致性的精确定义相当微妙，我们将在本节的剩余部分探讨它。但是基本的想法是让一个系统看起来好像只有一个数据副本，而且所有的操作都是原子性的。有了这个保证，即使实际中可能有多个副本，应用也不需要担心它们。

在一个线性一致的系统中，只要一个客户端成功完成写操作，所有客户端从数据库中读取数据必须能够看到刚刚写入的值。维护数据的单个副本的错觉是指，系统能保障读到的值是最近的，最新的，而不是来自陈旧的缓存或副本。换句话说，线性一致性是一个新鲜度保证（**recency guarantee**）。为了阐明这个想法，我们来看看一个非线性一致系统的例子。

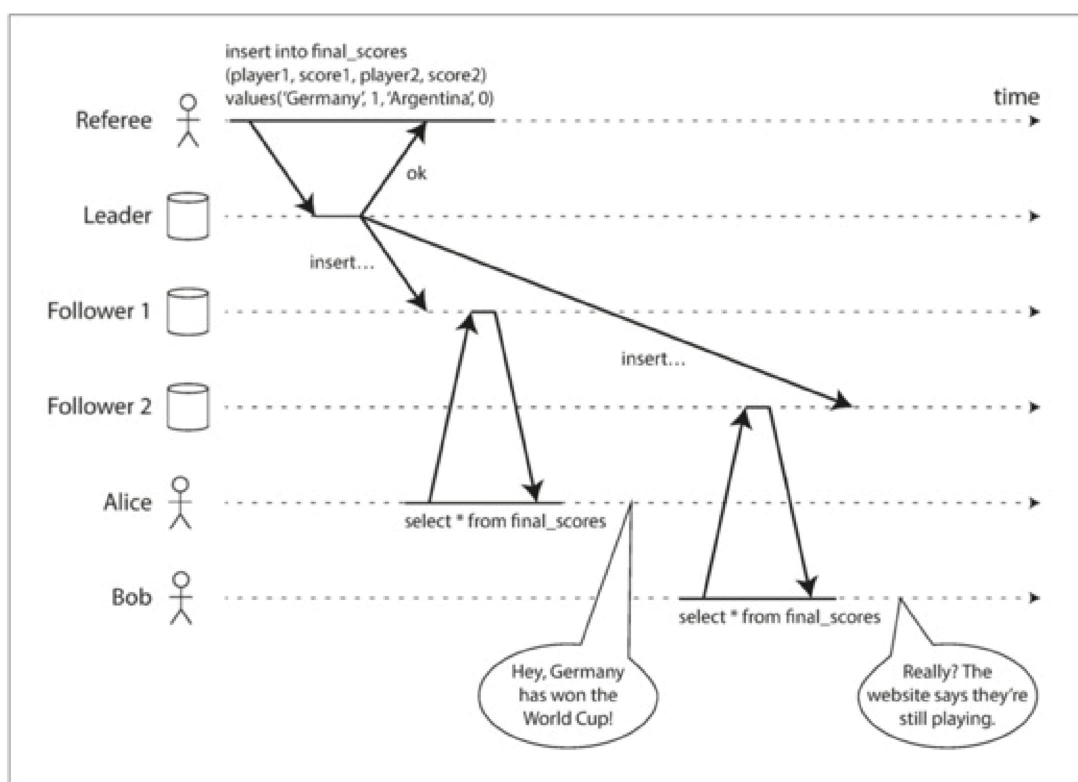


图**9-1** 这个系统是非线性一致的，导致了球迷的困惑

图9-1 展示了一个关于体育网站的非线性一致例子【9】。Alice和Bob正坐在同一个房间里，都盯着各自的手机，关注着2014年FIFA世界杯决赛的结果。在最后得分公布后，Alice刷新页面，看到宣布了获胜者，并兴奋地告诉Bob。Bob难以置信地刷新了自己的手机，但他的请求路由到了一个落后的数据库副本上，手机显示比赛仍在进行。

如果Alice和Bob在同一时间刷新并获得了两个不同的查询结果，也许就没有那么令人惊讶了。因为他们不知道服务器处理他们请求的精确时刻。然而Bob是在听到Alice惊呼最后得分之后，点击了刷新按钮（启动了他的查询），因此他希望查询结果至少与爱丽丝一样新鲜。但他的查询返回了陈旧结果，这一事实违背了线性一致性的要求。

什么使得系统线性一致？

线性一致性背后的基本思想很简单：使系统看起来好像只有一个数据副本。然而确切来讲，实际上有更多要操心的地方。为了更好地理解线性一致性，让我们再看几个例子。

图9-2 显示了三个客户端在线性一致数据库中同时读写相同的键 `x` 。在分布式系统文献中， `x` 被称为寄存器（**register**），例如，它可以是键值存储中的一个键，关系数据库中的一行，或文档数据库中的一个文档。

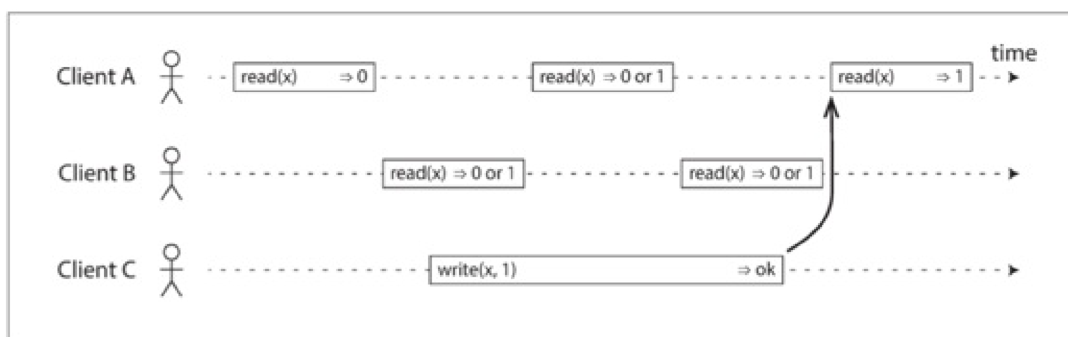


图**9-2** 如果读取请求与写入请求并发，则可能会返回旧值或新值

为了简单起见，图9-2采用了用户请求的视角，而不是数据库内部的视角。每个柱都是由客户端发出的请求，其中柱头是请求发送的时刻，柱尾是客户端收到响应的时刻。因为网络延迟变化无常，客户端不知道数据库处理其请求的精确时间——只知道它发生在发送请求和接收响应的之间的某个时刻。[i]

> i. 这个图的一个微妙的细节是它假定存在一个全局时钟，由水平轴表示。即使真实的系统通常没有准确的时钟（参阅“不可靠的时钟”），但这种假设是允许的：为了分析分布式算法，我们可以假设一个精确的全局时钟存在，不过算法无法访问它【47】。算法只能看到由石英振荡器和NTP产生的实时逼近。↩

在这个例子中，寄存器有两种类型的操作：

- $ read(x)⇒v$表示客户端请求读取寄存器 `x` 的值，数据库返回值 `v` 。
- $write(x,v)⇒r$ 表示客户端请求将寄存器 `x` 设置为值 `v` ，数据库返回响应 `r` （可能正确，可能错误）。

在图9-2 中， `x` 的值最初为 `0` ，客户端C 执行写请求将其设置为 `1` 。发生这种情况时，客户端A和B反复轮询数据库以读取最新值。 A和B的请求可能会收到怎样的响应？

- 客户端A的第一个读操作，完成于写操作开始之前，因此必须返回旧值 `0` 。
- 客户端A的最后一个读操作，开始于写操作完成之后。如果数据库是线性一致性的，它必然返回新值 `1` ：因为读操作和写操作一定是在其各自的起止区间内的某个时刻被处理。如果在写入结束后开始读取，则必须在写入之后处理读取，因此它必须看到写入的新值。
- 与写操作在时间上重叠的任何读操作，可能会返回 `0` 或 `1` ，因为我们不知道读取时，写操作是否已经生效。这些操作是并发（**concurrent**）的。

但是，这还不足以完全描述线性一致性：如果与写入同时发生的读取可以返回旧值或新值，那么读者可能会在写入期间看到数值在旧值和新值之间来回翻转。这不是我们所期望的仿真“单一数据副本”的系统。[ii]

> ii. 如果读取（与写入同时发生时）可能返回旧值或新值，则称该寄存器为常规寄存器（**regular register**）【7,25】↩

为了使系统线性一致，我们需要添加另一个约束，如图9-3所示

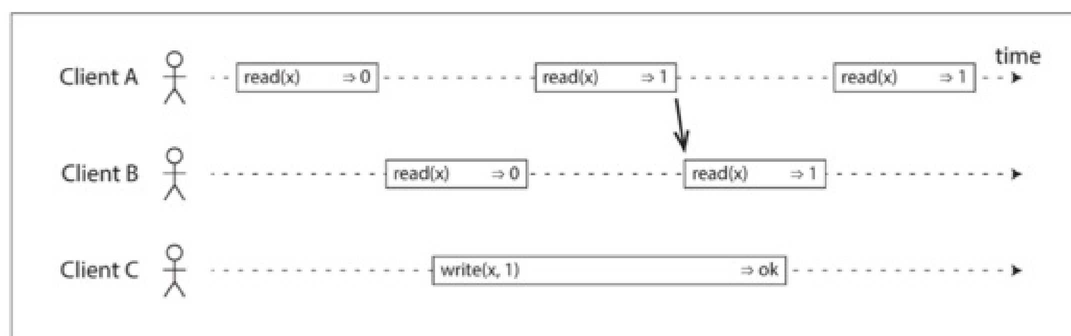


图**9-3** 任何一个读取返回新值后，所有后续读取（在相同或其他客户端上）也必须返回新值。

在一个线性一致的系统中，我们可以想象，在 `x` 的值从 `0` 自动翻转到 `1` 的时候（在写操作的开始和结束之间）必定有一个时间点。因此，如果一个客户端的读取返回新的值 `1`，即使写操作尚未完成，所有后续读取也必须返回新值。

图9-3中的箭头说明了这个时序依赖关系。客户端A 是第一个读取新的值 `1` 的位置。在A 的读取返回之后，B开始新的读取。由于B的读取严格在发生于A的读取之后，因此即使C的写入仍在进行中，也必须返回 `1`。（与图9-1中的Alice和Bob的情况相同：在Alice读取新值之后，Bob也希望读取新的值。）

我们可以进一步细化这个时序图，展示每个操作是如何在特定时刻原子性生效的。图9-4显示了一个更复杂的例子【10】。

在图9-4中，除了读写之外，还增加了第三种类型的操作：

- $cas(x, v*{old}*, v{new})⇒r$ 表示客户端请求进行原子性的比较与设置操作。如果寄存器 x 的当前值等于 $v*{old}*$ ，则应该原子地设置为 *$v*{new}$ 。如果 $x≠v_{old}$ ，则操作应该保持寄存器不变并返回一个错误。 r 是数据库的响应（正确或错误）。

图9-4中的每个操作都在我们认为执行操作的时候用竖线标出（在每个操作的条柱之内）。这些标记按顺序连在一起，其结果必须是一个有效的寄存器读写序列（每次读取都必须返回最近一次写入设置的值）。

线性一致性的要求是，操作标记的连线总是按时间（从左到右）向前移动，而不是向后移动。这个要求确保了我们之前讨论的新鲜性保证：一旦新的值被写入或读取，所有后续的读都会看到写入的值，直到它被再次覆盖。

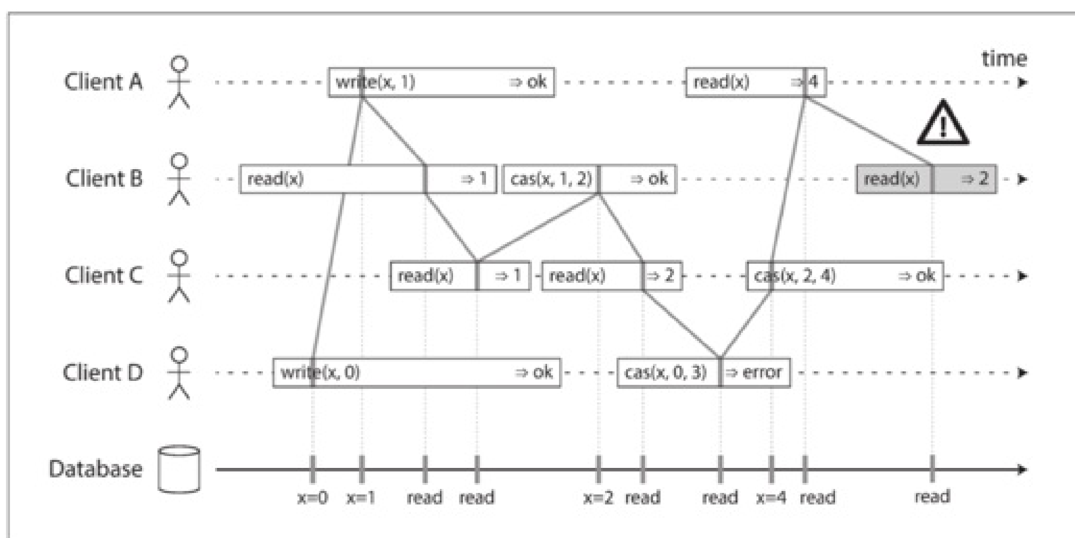


图**9-4** 可视化读取和写入看起来已经生效的时间点。 **B**的最后读取不是线性一致性的

图9-4中有一些有趣的细节需要指出：

- 第一个客户端B发送一个读取 `x` 的请求，然后客户端D发送一个请求将 `x` 设置为 `0`，然后客户端A发送请求将 `x` 设置为 `1`。尽管如此，返回到B的读取值为 `1`（由A写入的值）。这是可以的：这意味着数据库首先处理D的写入，然后是A的写入，最后是B的读取。虽然这不是请求发送的顺序，但这是一个可以接受的顺序，因为这三个请求是并发的。也许B的读请求在网络上略有延迟，所以它在两次写入之后才到达数据库。

- 在客户端A从数据库收到响应之前，客户端B的读取返回 `1` ，表示写入值 `1` 已成功。这也是可以的：这并不意味着在写之前读到了值，这只是意味着从数据库到客户端A的正确响应在网络中略有延迟。

- 此模型不假设有任何事务隔离：另一个客户端可能随时更改值。例如，C首先读取 `1` ，然后读取 `2` ，因为两次读取之间的值由B更改。可以使用原子比较并设置（**cas**）操作来检查该值是否未被另一客户端同时更改：B和C的**cas**请求成功，但是D的**cas**请求失败（在数据库处理它时， `x` 的值不再是 `0` ）。

- 客户B的最后一次读取（阴影条柱中）不是线性一致性的。 该操作与C的**cas**写操作并发（它将 `x` 从 `2` 更新为 `4` ）。在没有其他请求的情况下，B的读取返回 `2` 是可以的。然而，在B的读取开始之前，客户端A已经读取了新的值 `4` ，因此不允许B读取比A更旧的值。再次，与图9-1中的Alice和Bob的情况相同。

  这就是线性一致性背后的直觉。 正式的定义【6】更准确地描述了它。 通过记录所有请求和响应的时序，并检查它们是否可以排列成有效的顺序，测试一个系统的行为是否线性一致性是可能的（尽管在计算上是昂贵的）【11】。

> ## 线性一致性与可序列化
>
> 线性一致性容易和可序列化相混淆，因为两个词似乎都是类似“可以按顺序排列”的东西。但它们是两种完全不同的保证，区分两者非常重要：
>
> 可序列化
>
> 可序列化（Serializability）是事务的隔离属性，每个事务可以读写多个对象（行，文档，记录）——参阅“单对象和多对象操作”。它确保事务的行为，与它们按照某种顺序依次执行的结果相同（每个事务在下一个事务开始之前运行完成）。这种执行顺序可以与事务实际执行的顺序不同。【12】。
>
> 线性一致性
>
> 线性一致性（Linearizability）是读取和写入寄存器（单个对象）的新鲜度保证。它不会将操作组合为事务，因此它也不会阻止写偏差等问题（参阅“写偏差和幻读”），除非采取其他措施（例如物化冲突）。
>
> 一个数据库可以提供可串行性和线性一致性，这种组合被称为严格的可串行性或强的单副本强可串行性（strong-1SR）【4,13】。基于两阶段锁定的可串行化实现（参见“两阶段锁定（2PL）”一节）或实际串行执行（参见第“实际串行执行”）通常是线性一致性的。
>
> 但是，可序列化的快照隔离（参见“可序列化的快照隔离（SSI）”）不是线性一致性的：按照设计，它可以从一致的快照中进行读取，以避免锁定读者和写者之间的争用。一致性快照的要点就在于它不会包括比快照更新的写入，因此从快照读取不是线性一致性的。

依赖线性一致性

线性一致性在什么情况下有用？观看体育比赛的最后得分可能是一个轻率的例子：过了几秒钟的结果不可能在这种情况下造成任何真正的伤害。然而对于少数领域，线性一致性是系统正确工作的一个重要条件。

锁定和领导选举

一个使用单主复制的系统，需要确保领导真的只有一个，而不是几个（脑裂）。一种选择领导者的方法是使用锁：每个节点在启动时尝试获取锁，成功者成为领导者【14】。不管这个锁是如何实现的，它必须是线性一致的：所有节点必须就哪个节点拥有锁达成一致，否则就没用了。

诸如Apache ZooKeeper 【15】和etcd 【16】之类的协调服务通常用于实现分布式锁和领导者选举。它们使用一致性算法，以容错的方式实现线性一致的操作（在本章后面的“容错共识”中讨论此类算法）[iii]。还有许多微妙的细节来正确地实现锁和领导者选择（例如，参阅“领导者和锁”中的屏蔽问题），而像Apache Curator 【17】这样的库则通过在ZooKeeper之上提供更高级别的配方来提供帮助。但是，线性一致性存储服务是这些协调任务的基础。

iii

> iii. 严格地说，ZooKeeper和etcd提供线性一致性的写操作，但读取可能是陈旧的，因为默认情况下，它们可以由任何一个副本服务。你可以选择请求线性一致性读取：etcd调用这个法定读取【16】，而在ZooKeeper中，你需要在读取【15】之前调用`sync()`。参阅“使用全局顺序广播实现线性存储”。↩

分布式锁也在一些分布式数据库（如Oracle Real Application Clusters（RAC）【18】）中以更细的粒度使用。RAC对每个磁盘页面使用一个锁，多个节点共享对同一个磁盘存储系统的访问权限。由于这些线性一致的锁处于事务执行的关键路径上，RAC部署通常具有用于数据库节点之间通信的专用集群互连网络。

约束和唯一性保证

唯一性约束在数据库中很常见：例如，用户名或电子邮件地址必须唯一标识一个用户，而在文件存储服务中，不能有两个具有相同路径和文件名的文件。如果要在写入数据时强制执行此约束（例如，如果两个人试图同时创建一个具有相同名称的用户或文件，其中一个将返回一个错误），则需要线性一致性。

这种情况实际上类似于一个锁：当一个用户注册你的服务时，可以认为他们获得了所选用户名的“锁定”。该操作与原子性的比较与设置非常相似：将用户名赋予声明它的用户，前提是用户名尚未被使用。

如果想要确保银行账户余额永远不会为负数，或者不会出售比仓库里的库存更多的物品，或者两个人不会都预定了航班或剧院里同一时间的同一个位置。这些约束条件都要求所有节点都同意一个最新的值（账户余额，库存水平，座位占用率）。

在实际应用中，处理这些限制有时是可以接受的（例如，如果航班超额预订，你可以将客户转移到不同的航班并为其提供补偿）。在这种情况下，可能不需要线性一致性，我们将在“及时性与完整性”中讨论这种松散解释的约束。

然而，一个硬性的唯一性约束（关系型数据库中常见的那种）需要线性一致性。其他类型的约束，如外键或属性约束，可以在不需要线性一致性的情况下实现【19】。

跨信道的时序依赖

注意图9-1 中的一个细节：如果Alice没有惊呼得分，Bob就不会知道他的查询结果是陈旧的。他会在几秒钟之后再次刷新页面，并最终看到最后的分数。由于系统中存在额外的信道（Alice的声音传到了Bob的耳朵中），线性一致性的违背才被注意到。

计算机系统也会出现类似的情况。例如，假设有一个网站，用户可以上传照片，一个后台进程会调整照片大小，降低分辨率以加快下载速度（缩略图）。该系统的架构和数据流如图9-5所示。

图像缩放器需要明确的指令来执行尺寸缩放作业，指令是Web服务器通过消息队列发送的（参阅第11章）。 Web服务器不会将整个照片放在队列中，因为大多数消息代理都是针对较短的消息而设计的，而一张照片的空间占用可能达到几兆字节。取而代之的是，首先将照片

写入文件存储服务，写入完成后再将缩放器的指令放入消息队列。

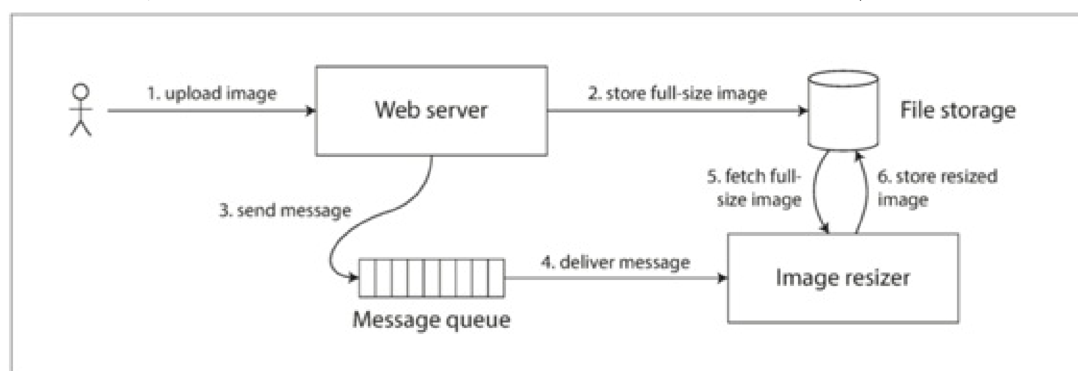


图9-5 Web服务器和图像调整器通过文件存储和消息队列进行通信，打开竞争条件的可能性。

如果文件存储服务是线性一致的，那么这个系统应该可以正常工作。如果它不是线性一致的，则存在竞争条件的风险：消息队列（图9-5中的步骤3和4）可能比存储服务内部的复制更快。在这种情况下，当缩放器读取图像（步骤5）时，可能会看到图像的旧版本，或者什么都没有。如果它处理的是旧版本的图像，则文件存储中的全尺寸图和略缩图就产生了永久性的不一致。

出现这个问题是因为Web服务器和缩放器之间存在两个不同的信道：文件存储与消息队列。没有线性一致性的新鲜性保证，这两个信道之间的竞争条件是可能的。这种情况类似于图9-1，数据库复制与Alice的嘴到Bob耳朵之间的真人音频信道之间也存在竞争条件。

线性一致性并不是避免这种竞争条件的唯一方法，但它是最容易理解的。如果你可以控制额外信道（例如消息队列的例子，而不是在Alice和Bob的例子），则可以使用在“读己之写”讨论过的备选方法，不过会有额外的复杂度代价。

实现线性一致的系统

我们已经见到了几个线性一致性有用的例子，让我们思考一下，如何实现一个提供线性一致语义的系统。

由于线性一致性本质上意味着“表现得好像只有一个数据副本，而且所有的操作都是原子的”，所以最简单的答案就是，真的只用一个数据副本。但是这种方法无法容错：如果持有该副本的节点失效，数据将会丢失，或者至少无法访问，直到节点重新启动。

使系统容错最常用的方法是使用复制。我们再来回顾第5章中的复制方法，并比较它们是否可以满足线性一致性：

单主复制（可能线性一致）

在具有单主复制功能的系统中（参见“领导者与追随者”），主库具有用于写入的数据的主副本，而追随者在其他节点上保留数据的备份副本。如果从主库或同步更新的从库读取数据，它们可能（**protential**）是线性一致性的[iv]。然而，并不是每个单主数据库都是实际线性一致性的，无论是通过设计（例如，因为使用快照隔离）还是并发错误【10】。

iv

> iv. 对单领域数据库进行分区（分片），以便每个分区有一个单独的领导者，不会影响线性一致性，因为线性一致性只是对单一对象的保证。 交叉分区事务是一个不同的问题（参阅“分布式事务和共识”）。 ↩

从主库读取依赖一个假设，你确定领导是谁。正如在“真理在多数人手中”中所讨论的那样，一个节点很可能会认为它是领导者，而事实上并非如此——如果具有错觉的领导者继续为请求提供服务，可能违反线性一致性【20】。使用异步复制，故障转移时甚至可能会丢失已提交的写入（参阅“处理节点宕机”），这同时违反了持久性和线性一致性。

共识算法（线性一致）

一些在本章后面讨论的共识算法，与单领导者复制类似。然而，共识协议包含防止脑裂和陈旧副本的措施。由于这些细节，共识算法可以安全地实现线性一致性存储。例如，Zookeeper【21】和etcd 【22】就是这样工作的。

多主复制（非线性一致）

具有多主程序复制的系统通常不是线性一致的，因为它们同时在多个节点上处理写入，并将其异步复制到其他节点。因此，它们可能会产生冲突的写入，需要解析（参阅“处理写入冲突”）。这种冲突是因为缺少单一数据副本人为产生的。

无主复制（也许不是线性一致的）

对于无领导者复制的系统（Dynamo风格；参阅“无主复制”），有时候人们会声称通过要求法定人数读写（ $w + r> n$ ）可以获得“强一致性”。这取决于法定人数的具体配置，以及强一致性如何定义（通常不完全正确）。

基于时钟（例如，在Cassandra中；参见“依赖同步时钟”）的“最后写入胜利”冲突解决方法几乎可以确定是非线性的，由于时钟偏差，不能保证时钟的时间戳与实际事件顺序一致。松散的法定人数也破坏了线性一致的可能性。即使使用严格的法定人数，非线性一致的行为也是可能的，如下节所示。

线性一致性和法定人数

直觉上在Dynamo风格的模型中，严格的法定人数读写应该是线性一致性的。但是当我们有可变的网络延迟时，就可能存在竞争条件，如图9-6所示。

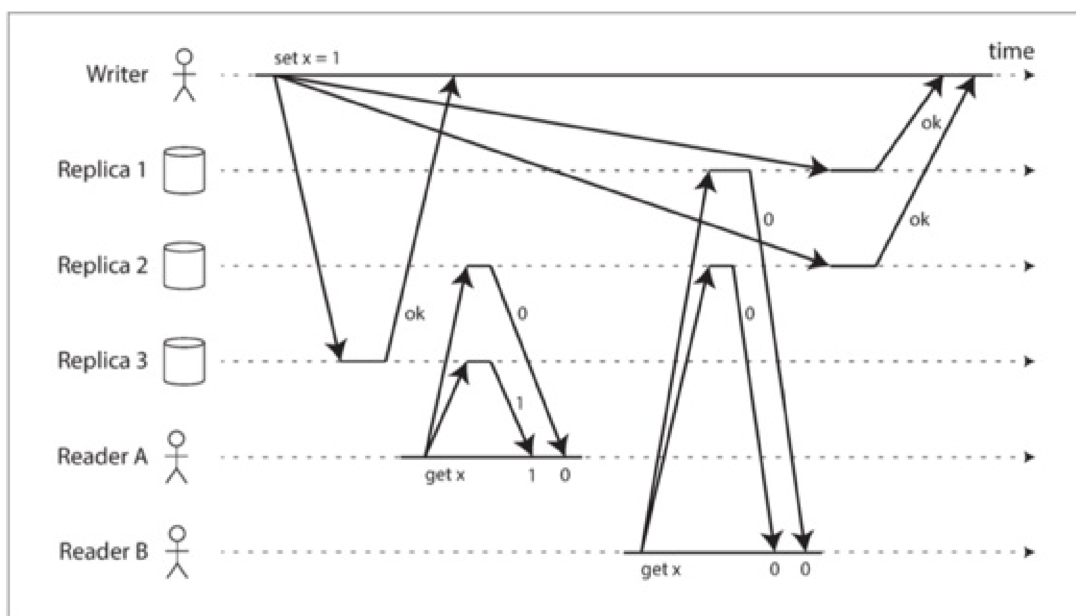


图**9-6** 非线性一致的执行，尽管使用了严格的法定人数

在图9-6中，x 的初始值为0，写入客户端通过向所有三个副本（ $n = 3, w = 3$ ）发送写入将 x 更新为 `1` 。客户端A并发地从两个节点组成的法定人群（ $r = 2$ ）中读取数据，并在其中一个节点上看到新值 `1` 。客户端B也并发地从两个不同的节点组成的法定人数中读取，并从两个节点中取回了旧值 `0` 。

仲裁条件满足（ $w + r> n$ ），但是这个执行是非线性一致的：B的请求在A的请求完成后开始，但是B返回旧值，而A返回新值。（又一次，如同Alice和Bob的例子 图9-1）

有趣的是，通过牺牲性能，可以使Dynamo风格的法定人数线性化：读取者必须在将结果返回给应用程序之前，同步执行读取修复（参阅“读时修复与反熵过程”），并且写入者必须在发送写入之前，读取法定数量节点的最新状态【24,25】。然而，由于性能损失，Riak不执行同步读取修复【26】。 Cassandra在进行法定人数读取时，确实在等待读取修复完成【27】；但是由于使用了最后写入为准的冲突解决方案，当同一个键有多个并发写入时，将不能保证线性一致性。

而且，这种方式只能实现线性一致的读写；不能实现线性一致的比较和设置操作，因为它需要一个共识算法【28】。

总而言之，最安全的做法是：假设采用Dynamo风格无主复制的系统不能提供线性一致性。

线性一致性的代价

一些复制方法可以提供线性一致性，另一些复制方法则不能，因此深入地探讨线性一致性的优缺点是很有趣的。

我们已经在第五章中讨论了不同复制方法的一些用例。例如对多数据中心的复制而言，多主复制通常是理想的选择（参阅“运维多个数据中心”）。图9-7说明了这种部署的一个例子。

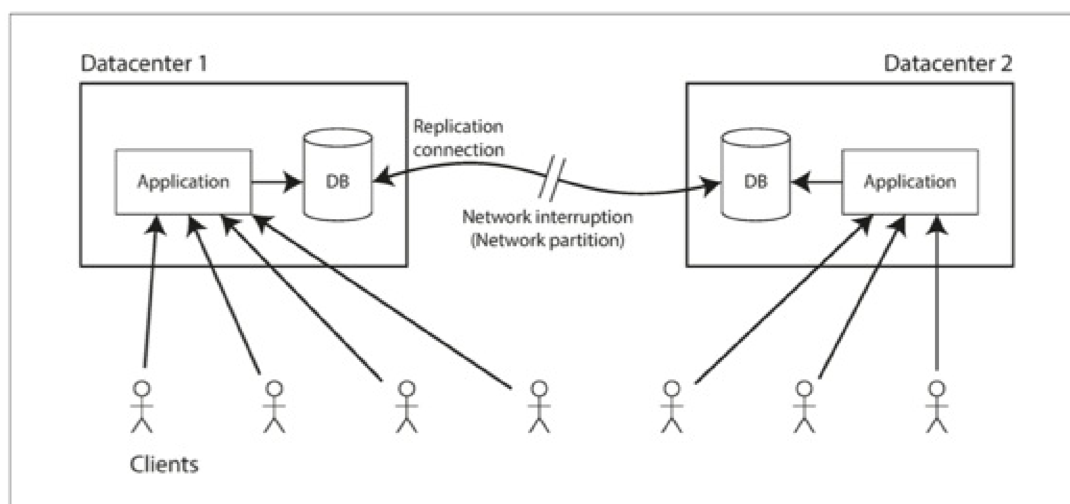


图9-7 网络中断迫使在线性一致性和可用性之间做出选择。

考虑这样一种情况：如果两个数据中心之间发生网络中断会发生什么？我们假设每个数据中心内的网络正在工作，客户端可以访问数据中心，但数据中心之间彼此无法互相连接。

使用多主数据库，每个数据中心都可以继续正常运行：由于在一个数据中心写入的数据是异步复制到另一个数据中心的，所以在恢复网络连接时，写入操作只是简单地排队并交换。

另一方面，如果使用单主复制，则主库必须位于其中一个数据中心。任何写入和任何线性一致的读取请求都必须发送给该主库，因此对于连接到从库所在数据中心的客户端，这些读取和写入请求必须通过网络同步发送到主库所在的数据中心。

在单主配置的条件下，如果数据中心之间的网络被中断，则连接到从库数据中心的客户端无法联系到主库，因此它们无法对数据库执行任何写入，也不能执行任何线性一致的读取。它们仍能从从库读取，但结果可能是陈旧的（非线性一致）。如果应用需要线性一致的读写，却又位于与主库网络中断的数据中心，则网络中断将导致这些应用不可用。

如果客户端可以直接连接到主库所在的数据中心，这就不是问题了，哪些应用可以继续正常工作。但直到网络链接修复之前，只能访问从库数据中心的客户端会中断运行。

CAP定理

这个问题不仅仅是单主复制和多主复制的后果：任何线性一致的数据库都有这个问题，不管它是如何实现的。这个问题也不仅仅局限于多数据中心部署，而可能发生在任何不可靠的网络上，即使在同一个数据中心内也是如此。问题面临的权衡如下：[v]

- 如果应用需要线性一致性，且某些副本因为网络问题与其他副本断开连接，那么这些副本掉线时不能处理请求。请求必须等到网络问题解决，或直接返回错误。（无论哪种方式，服务都不可用（**unavailable**））。
- 如果应用不需要线性一致性，那么某个副本即使与其他副本断开连接，也可以独立处理请求（例如多主复制）。在这种情况下，应用可以在网络问题前保持可用，但其行为不是线性一致的。

v

> v. 这两种选择有时分别称为CP（在网络分区下一致但不可用）和AP（在网络分区下可用但不一致）。但是，这种分类方案存在一些缺陷【9】，所以最好不要这样用。↩

因此不需要线性一致性的应用对网络问题有更强的容错能力。这种见解通常被称为CAP定理【29,30,31,32】，由Eric Brewer于2000年命名，尽管70年代的分布式数据库设计者早就知道了这种权衡【33,34,35,36】。

CAP最初是作为一个经验法则提出的，没有准确的定义，目的是开始讨论数据库的权衡。那时候许多分布式数据库侧重于在共享存储的集群上提供线性一致性的语义【18】，CAP定理鼓励数据库工程师向分布式无共享系统的设计领域深入探索，这类架构更适合实现大规模的网络服务【37】。对于这种文化上的转变，CAP值得赞扬 —— 它见证了自00年代中期以来新数据库的技术爆炸（即NoSQL）。

> ## CAP定理没有帮助
>
> CAP有时以这种面目出现：一致性，可用性和分区容忍：三者只能择其二。不幸的是这种说法很有误导性【32】，因为网络分区是一种错误，所以它并不是一个选项：不管你喜不喜欢它都会发生【38】。
>
> 在网络正常工作的时候，系统可以提供一致性（线性一致性）和整体可用性。发生网络故障时，你必须在线性一致性和整体可用性之间做出选择。因此，一个更好的表达CAP的方法可以是一致的，或者在分区时可用【39】。一个更可靠的网络需要减少这个选择，但是在某些时候选择是不可避免的。
>
> 在CAP的讨论中，术语可用性有几个相互矛盾的定义，形式化作为一个定理【30】并不符合其通常的含义【40】。许多所谓的“高可用”（容错）系统实际上不符合CAP对可用性的特殊定义。总而言之，围绕着CAP有很多误解和困惑，并不能帮助我们更好地理解系统，所以最好避免使用CAP。

CAP定理的正式定义仅限于很狭隘的范围【30】，它只考虑了一个一致性模型（即线性一致性）和一种故障（网络分区[vi]，或活跃但彼此断开的节点）。它没有讨论任何关于网络延迟，死亡节点或其他权衡的事。因此，尽管CAP在历史上有一些影响力，但对于设计系统而言并没有实际价值【9,40】。

在分布式系统中有更多有趣的“不可能”的结果【41】，且CAP定理现在已经被更精确的结果取代【2,42】，所以它现在基本上成了历史古迹了。

> vi. 正如“真实世界的网络故障”中所讨论的，本书使用分区（**partition**）指代将大数据集细分为小数据集的操作（分片；参见第6章）。与之对应的是，网络分区（**network partition**）是一种特定类型的网络故障，我们通常不会将其与其他类型的故障分开考虑。但是，由于它是CAP的P，所以这种情况下不能将其混为一谈。↩

线性一致性和网络延迟

虽然线性一致是一个很有用的保证，但实际上，线性一致的系统惊人的少。例如，现代多核CPU上的内存甚至都不是线性一致的【43】：如果一个CPU核上运行的线程写入某个内存地址，而另一个CPU核上运行的线程不久之后读取相同的地址，并没有保证一定能一定读到第一个线程写入的值（除非使用了内存屏障（**memory barrier**）或围栏（**fence**）【44】）。

这种行为的原因是每个CPU核都有自己的内存缓存和存储缓冲区。默认情况下，内存访问首先走缓存，任何变更会异步写入主存。因为缓存访问比主存要快得多【45】，所以这个特性对于现代CPU的良好性能表现至关重要。但是现在就有几个数据副本（一个在主存中，也许还有几个在不同缓存中的其他副本），而且这些副本是异步更新的，所以就失去了线性一致性。

为什么要做这个权衡？对多核内存一致性模型而言，CAP定理是没有意义的：在同一台计算机中，我们通常假定通信都是可靠的。并且我们并不指望一个CPU核能在脱离计算机其他部分的条件下继续正常工作。牺牲线性一致性的原因是性能（**performance**），而不是容错。

许多分布式数据库也是如此：它们是为了提高性能而选择了牺牲线性一致性，而不是为了容错【46】。线性一致的速度很慢——这始终是事实，而不仅仅是网络故障期间。

能找到一个更高效的线性一致存储实现吗？看起来答案是否定的：Attiya和Welch 【47】证明，如果你想要线性一致性，读写请求的响应时间至少与网络延迟的不确定性成正比。在像大多数计算机网络一样具有高度可变延迟的网络中（参见“超时与无穷的延迟”），线性读写的响应时间不可避免地会很高。更快地线性一致算法不存在，但更弱的一致性模型可以快得多，所以对延迟敏感的系统而言，这类权衡非常重要。在第12章中将讨论一些在不牺牲正确性的前提下，绕开线性一致性的方法。

顺序保证

之前说过，线性一致寄存器的行为就好像只有单个数据副本一样，且每个操作似乎都是在某个时间点以原子性的方式生效的。这个定义意味着操作是按照某种良好定义的顺序执行的。我们通过操作（似乎）执行完毕的顺序来连接操作，以此说明图9-4中的顺序。

顺序（**ordering**）这一主题在本书中反复出现，这表明它可能是一个重要的基础性概念。让我们简要回顾一下其它顺序曾经出现过的上下文：

- 在第5章中我们看到，领导者在单主复制中的主要目的就是，在复制日志中确定写入顺序（**order of write**）——也就是从库应用这些写入的顺序。如果不存在一个领导者，则并发操作可能导致冲突（参阅“处理写入冲突”）。
- 在第7章中讨论的可序列化，是关于事务表现的像按某种序列顺序（**some sequential order**）执行的保证。它可以通过字面意义上地序列顺序（**serial order**）执行事务来实现，或者通过允许并行执行，同时防止序列化冲突来实现（通过锁或中止事务）。
- 在第8章讨论过的在分布式系统中使用时间戳和时钟（参阅“依赖于同步时钟”）是另一种将顺序引入无序世界的尝试，例如，确定两个写入操作哪一个更晚发生。

事实证明，顺序，线性一致性和共识之间有着深刻的联系。尽管这个概念比本书其他部分更加理论化和抽象，但对于明确系统的能力范围（可以做什么和不可以做什么）而言是非常有帮助的。我们将在接下来的几节中探讨这个话题。

顺序与因果

顺序反复出现有几个原因，其中一个原因是，它有助于保持因果关系（**causality**）。在本书中我们已经看到了几个例子，其中因果关系是很重要的：

- 在“一致前缀读”（图5-5）中，我们看到一个例子：一个对话的观察者首先看到问题的答案，然后才看到被回答的问题。这是令人困惑的，因为它违背了我们对因（**cause**）与果（**effect**）的直觉：如果一个问题被回答，显然问题本身得先在那里，因为给出答案的人必须看到这个问题（假如他们并没有预见未来的超能力）。我们认为在问题和答案之间存在因果依赖（**causal dependency**）。
- 图5-9中出现了类似的模式，我们看到三位领导者之间的复制，并注意到由于网络延迟，一些写入可能会“压倒”其他写入。从其中一个副本的角度来看，好像有一个对尚不存在的记录的更新操作。这里的因果意味着，一条记录必须先被创建，然后才能被更新。
- 在“检测并发写入”中我们观察到，如果有两个操作A和B，则存在三种可能性：A发生在B之前，或B发生在A之前，或者A和B并发。这种此前发生（**happened before**）关系是因果关系的另一种表述：如果A在B前发生，那么意味着B可能已经知道了A，或者建立在A的基础上，或者依赖于A。如果A和B是并发的，那么它们之间并没有因果联系；换句话说，我们确信A和B不知道彼此。
- 在事务快照隔离的上下文中（“快照隔离和可重复读”），我们说事务是从一致性快照中读取的。但此语境中“一致”到底又是什么意思？这意味着与因果关系保持一致（**consistent with causality**）：如果快照包含答案，它也必须包含被回答的问题【48】。在某个时间点观察整个数据库，与因果关系保持一致意味着：因果上在该时间点之前发生的所有操作，其影响都是可见的，但因果上在该时间点之后发生的操作，其影响对观察者不可见。读偏差（**read skew**）意味着读取的数据处于违反因果关系的状态（不可重复读，如图7-6所示）。
- 事务之间写偏差（**write skew**）的例子（参见“写偏差和幻象”）也说明了因果依赖：在图7-8中，爱丽丝被允许离班，因为事务认为鲍勃仍在值班，反之亦然。在这种情况下，离班的动作因果依赖于对当前值班情况的观察。可序列化的快照隔离通过跟踪事务之间的因果依赖来检测写偏差。
- 在爱丽丝和鲍勃看球的例子中（图9-1），在听到爱丽丝惊呼比赛结果后，鲍勃从服务器得到陈旧结果的事实违背了因果关系：爱丽丝的惊呼因果依赖于得分宣告，所以鲍勃应该也能在听到爱丽斯惊呼后查询到比分。相同的模式在“跨信道的时序依赖”一节中，以“图像大小调整服务”的伪装再次出现。

因果关系对事件施加了一种顺序：因在果之前；消息发送在消息收取之前。而且就像现实生活中一样，一件事会导致另一件事：某个节点读取了一些数据然后写入一些结果，另一个节点读取其写入的内容，并依次写入一些其他内容，等等。这些因果依赖的操作链定义了系统

中的因果顺序，即，什么在什么之前发生。

如果一个系统服从因果关系所规定的顺序，我们说它是因果一致（**causally**）的。例如，快照隔离提供了因果一致性：当你从数据库中读取到一些数据时，你一定还能够看到其因果前驱（假设在此期间这些数据还没有被删除）。

因果顺序不是全序的

全序（**total order**）允许任意两个元素进行比较，所以如果有两个元素，你总是可以说出哪个更大，哪个更小。例如，自然数集是全序的：给定两个自然数，比如说5和13，那么你可以告诉我，13大于5。

然而数学集合并不完全是全序的：`{a, b}` 比 `{b, c}` 更大吗？好吧，你没法真正比较它们，因为二者都不是对方的子集。我们说它们是无法比较（**incomparable**）的，因此数学集合是偏序（**partially order**）的：在某些情况下，可以说一个集合大于另一个（如果一个集合包含另一个集合的所有元素），但在其他情况下它们是无法比较的[译注i]。

> 译注i. 设R为非空集合A上的关系，如果R是自反的、反对称的和可传递的，则称R为A上的偏序关系。简称偏序，通常记作≦。一个集合A与A上的偏序关系R一起叫作偏序集，记作(A,R)或$(A, ≦)$。全序、偏序、关系、集合，这些概念的精确定义可以参考任意一本离散数学教材。↩

全序和偏序之间的差异反映在不同的数据库一致性模型中：

线性一致性

在线性一致的系统中，操作是全序的：如果系统表现的就好像只有一个数据副本，并且所有操作都是原子性的，这意味着对任何两个操作，我们总是能判定哪个操作先发生。这个全序图9-4中以时间线表示。

因果性

我们说过，如果两个操作都没有在彼此之前发生，那么这两个操作是并发的（参阅“此前发生”的关系和并发）。换句话说，如果两个事件是因果相关的（一个发生在另一个事件之前），则它们之间是有序的，但如果它们是并发的，则它们之间的顺序是无法比较的。这意味着因果关系定义了一个偏序，而不是一个全序：一些操作相互之间是有顺序的，但有些则是无法比较的。

因此，根据这个定义，在线性一致的数据存储中是不存在并发操作的：必须有且仅有一条时间线，所有的操作都在这条时间线上，构成一个全序关系。可能有几个请求在等待处理，但是数据存储确保了每个请求都是在唯一时间线上的某个时间点自动处理的，不存在任何并发。

并发意味着时间线会分岔然后合并 —— 在这种情况下，不同分支上的操作是无法比较的（即并发操作）。在第五章中我们看到了这种现象：例如，图5-14 并不是一条直线的全序关系，而是一堆不同的操作并发进行。图中的箭头指明了因果依赖 —— 操作的偏序。

如果你熟悉像Git这样的分布式版本控制系统，那么其版本历史与因果关系图极其相似。通常，一个提交（**Commit**）发生在另一个提交之后，在一条直线上。但是有时你会遇到分支（当多个人同时在一个项目上工作时），合并（**Merge**）会在这些并发创建的提交相融合时创建。

线性一致性强于因果一致性

那么因果顺序和线性一致性之间的关系是什么？答案是线性一致性隐含着（**implies**）因果关系：任何线性一致的系统都能正确保持因果性【7】。特别是，如果系统中有多个通信通道（如图9-5 中的消息队列和文件存储服务），线性一致性可以自动保证因果性，系统无需任何特殊操作（如在不同组件间传递时间戳）。

线性一致性确保因果性的事实使线性一致系统变得简单易懂，更有吸引力。然而，正如“线性一致性的代价”中所讨论的，使系统线性一致可能会损害其性能和可用性，尤其是在系统具有严重的网络延迟的情况下（例如，如果系统在地理上散布）。出于这个原因，一些分布式数据系统已经放弃了线性一致性，从而获得更好的性能，但它们用起来也更为困难。

好消息是存在折衷的可能性。线性一致性并不是保持因果性的唯一途径 —— 还有其他方法。一个系统可以是因果一致的，而无需承担线性一致带来的性能折损（尤其对于CAP定理不适用的情况）。实际上在所有的不会被网络延迟拖慢的一致性模型中，因果一致性是可行的最强的一致性模型。而且在网络故障时仍能保持可用【2,42】。

在许多情况下，看上去需要线性一致性的系统，实际上需要的只是因果一致性，因果一致性可以更高效地实现。基于这种观察结果，研究人员正在探索新型的数据库，既能保证因果一致性，且性能与可用性与最终一致的系统类似【49,50,51】。

这方面的研究相当新鲜，其中很多尚未应用到生产系统，仍然有不少挑战需要克服【52,53】。但对于未来的系统而言，这是一个有前景的方向。

捕获因果关系

我们不会在这里讨论非线性一致的系统如何保证因果性的细节，而只是简要地探讨一些关键的思想。

为了维持因果性，你需要知道哪个操作发生在哪个其他操作之前（**happened before**）。这是一个偏序：并发操作可以以任意顺序进行，但如果一个操作发生在另一个操作之前，那它们必须在所有副本上以那个顺序被处理。因此，当一个副本处理一个操作时，它必须确保所有因果前驱的操作（之前发生的所有操作）已经被处理；如果前面的某个操作丢失了，后面的操作必须等待，直到前面的操作被处理完毕。

为了确定因果依赖，我们需要一些方法来描述系统中节点的“知识”。如果节点在发出写入Y 的请求时已经看到了 X的值，则 X 和 Y 可能存在因果关系。这个分析使用了那些在欺诈指控刑事调查中常见的问题：CEO在做出决定 Y 时是否知道 X ？

用于确定哪些操作发生在其他操作之前 的技术，与我们在“检测并发写入”中所讨论的内容类似。那一节讨论了无领导者数据存储中的因果性：为了防止丢失更新，我们需要检测到对同一个键的并发写入。因果一致性则更进一步：它需要跟踪整个数据库中的因果依赖，而不仅仅是一个键。可以推广版本向量以解决此类问题【54】。

为了确定因果顺序，数据库需要知道应用读取了哪个版本的数据。这就是为什么在 图5-13 中，来自先前操作的版本号在写入时被传回到数据库的原因。在SSI 的冲突检测中会出现类似的想法，如“可序列化的快照隔离（SSI）”中所述：当事务要提交时，数据库将检查它所读取的数据版本是否仍然是最新的。为此，数据库跟踪哪些数据被哪些事务所读取。

序列号顺序

虽然因果是一个重要的理论概念，但实际上跟踪所有的因果关系是不切实际的。在许多应用中，客户端在写入内容之前会先读取大量数据，我们无法弄清写入因果依赖于先前全部的读取内容，还是仅包括其中一部分。显式跟踪所有已读数据意味着巨大的额外开销。

但还有一个更好的方法：我们可以使用序列号（**sequence nunber**）或时间戳（**timestamp**）来排序事件。时间戳不一定来自时钟（或物理时钟，存在许多问题，如 “不可靠时钟” 中所述）。它可以来自一个逻辑时钟（**logical clock**），这是一个用来生成标识操作的数字序列的算法，典型实现是使用一个每次操作自增的计数器。

这样的序列号或时间戳是紧凑的（只有几个字节大小），它提供了一个全序关系：也就是说每操作都有一个唯一的序列号，而且总是可以比较两个序列号，确定哪一个更大（即哪些操作后发生）。

特别是，我们可以使用与因果一致（**consistent with causality**）的全序来生成序列号[vii]：我们保证，如果操作 A 因果后继于操作 B，那么在这个全序中 A 在 B 前（ A 具有比 B 更小的序列号）。并行操作之间可以任意排序。这样一个全序关系捕获了所有关于因果的信息，但也施加了一个比因果性要求更为严格的顺序。

> [vii]. 与因果关系不一致的全序很容易创建，但没啥用。例如你可以为每个操作生成随机的UUID，并按照字典序比较UUID，以定义操作的全序。这是一个有效的全序，但是随机的UUID并不能告诉你哪个操作先发生，或者操作是否为并发的。↩

在单主复制的数据库中（参见“领导者与追随者”），复制日志定义了与因果一致的写操作。主库可以简单地为每个操作自增一个计数器，从而为复制日志中的每个操作分配一个单调递增的序列号。如果一个从库按照它们在复制日志中出现的顺序来应用写操作，那么从库的状态始终是因果一致的（即使它落后于领导者）。

非因果序列号生成器

如果主库不存在（可能因为使用了多主数据库或无主数据库，或者因为使用了分区的数据库），如何为操作生成序列号就没有那么明显了。在实践中有各种各样的方法：

- 每个节点都可以生成自己独立的一组序列号。例如有两个节点，一个节点只能生成奇数，而另一个节点只能生成偶数。通常，可以在序列号的二进制表示中预留一些位，用于唯一的节点标识符，这样可以确保两个不同的节点永远不会生成相同的序列号。
- 可以将时钟（物理时钟）时间戳附加到每个操作上【55】。这种时间戳并不连续，但是如果它具有足够高的分辨率，那也许足以提供一个操作的全序关系。这一事实应用于 最后写入为准 的冲突解决方法中（参阅“有序事件的时间戳”）。
- 可以预先分配序列号区块。例如，节点 A 可能要求从序列号1到1,000区块的所有权，而节点 B 可能要求序列号1,001到2,000区块的所有权。然后每个节点可以独立分配所属区块中的序列号，并在序列号告急时请求分配一个新的区块。

这三个选项都比单一主库的自增计数器表现要好，并且更具可扩展性。它们为每个操作生成一个唯一的，近似自增的序列号。然而它们都有同一个问题：生成的序列号与因果不一致。

因为这些序列号生成器不能正确地捕获跨节点的操作顺序，所以会出现因果关系的问题：

- 每个节点每秒可以处理不同数量的操作。因此，如果一个节点产生偶数序列号而另一个产生奇数序列号，则偶数计数器可能落后于奇数计数器，反之亦然。如果你有一个奇数编号的操作和一个偶数编号的操作，你无法准确地说出哪一个操作在因果上先发生。

- 来自物理时钟的时间戳会受到时钟偏移的影响，这可能会使其与因果不一致。例如图8-3展示了一个例子，其中因果上晚发生的操作，却被分配了一个更早的时间戳。[vii]

> [viii]. 可以使物理时钟时间戳与因果关系保持一致：在“用于全局快照的同步时钟”中，我们讨论了Google的Spanner，它可以估计预期的时钟偏差，并在提交写入之前等待不确定性间隔。 这中方法确保了实际上靠后的事务会有更大的时间戳。 但是大多数时钟不能提供这种所需的不确定性度量。 ↩

- 在分配区块的情况下，某个操作可能会被赋予一个范围在1,001到2,000内的序列号，然而一个因果上更晚的操作可能被赋予一个范围在1到1,000之间的数字。这里序列号与因果关系也是不一致的。

兰伯特时间戳

尽管刚才描述的三个序列号生成器与因果不一致，但实际上有一个简单的方法来产生与因果关系一致的序列号。它被称为兰伯特时间戳，莱斯利·兰伯特（Leslie Lamport）于1978年提出【56】，现在是分布式系统领域中被引用最多的论文之一。

图9-8 说明了兰伯特时间戳的应用。每个节点都有一个唯一标识符，和一个保存自己执行操作数量的计数器。 兰伯特时间戳就是两者的简单组合：（计数器，节点ID）$(counter, node ID)$。两个节点有时可能具有相同的计数器值，但通过在时间戳中包含节点ID，每个时间戳都是唯一的。

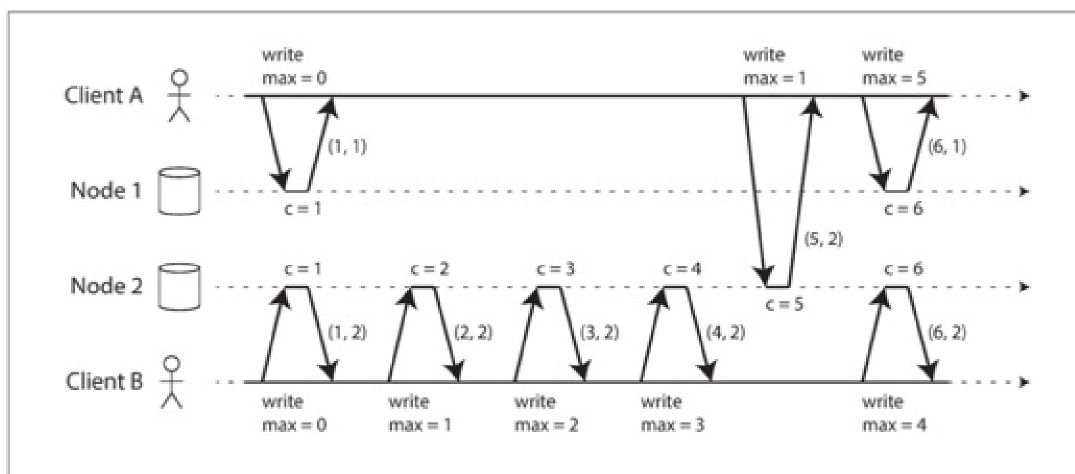


图**9-8 Lamport**时间戳提供了与因果关系一致的总排序。

兰伯特时间戳与物理时间时钟没有任何关系，但是它提供了一个全序：如果你有两个时间戳，则计数器值大者是更大的时间戳。如果计数器值相同，则节点ID越大的，时间戳越大。

迄今，这个描述与上节所述的奇偶计数器基本类似。使兰伯特时间戳因果一致的关键思想如下所示：每个节点和每个客户端跟踪迄今为止所见到的最大计数器值，并在每个请求中包含这个最大计数器值。当一个节点收到最大计数器值大于自身计数器值的请求或响应时，它立即将自己的计数器设置为这个最大值。

这如 图9-8 所示，其中客户端 A 从节点2 接收计数器值 `5` ，然后将最大值 `5` 发送到节点1 。此时，节点1 的计数器仅为 `1` ，但是它立即前移至 `5` ，所以下一个操作的计数器的值为 `6` 。

只要每一个操作都携带着最大计数器值，这个方案确保兰伯特时间戳的排序与因果一致，因为每个因果依赖都会导致时间戳增长。

兰伯特时间戳有时会与我们在 “检测并发写入” 中看到的版本向量相混淆。虽然两者有一些相似之处，但它们有着不同的目的：版本向量可以区分两个操作是并发的，还是一个因果依赖另一个；而兰伯特时间戳总是施行一个全序。从兰伯特时间戳的全序中，你无法分辨两个操作是并发的还是因果依赖的。 兰伯特时间戳优于版本向量的地方是，它更加紧凑。

光有时间戳排序还不够

虽然兰伯特时间戳定义了一个与因果一致的全序，但它还不足以解决分布式系统中的许多常见问题。

例如，考虑一个需要确保用户名能唯一标识用户帐户的系统。如果两个用户同时尝试使用相同的用户名创建帐户，则其中一个应该成功，另一个应该失败。 （我们之前在“领导者与锁定”中提到过这个问题。）

乍看之下，似乎操作的全序关系足以解决这一问题（例如使用兰伯特时间戳）：如果创建了两个具有相同用户名的帐户，选择时间戳较小的那个作为胜者（第一个抓到用户名的人），并让带有更大时间戳者失败。由于时间戳上有全序关系，所以这个比较总是可行的。

这种方法适用于事后确定胜利者：一旦你收集了系统中的所有用户名创建操作，就可以比较它们的时间戳。然而当某个节点需要实时处理用户创建用户名的请求时，这样的方法就无法满足了。节点需要马上（**right now**）决定这个请求是成功还是失败。在那个时刻，节点并不知道是否存其他节点正在并发执行创建同样用户名的操作，罔论其它节点可能分配给那个操作的时间戳。

为了确保没有其他节点正在使用相同的用户名和较小的时间戳并发创建同名账户，你必须检查其它每个节点，看看它在做什么【56】。如果其中一个节点由于网络问题出现故障或不可达，则整个系统可能被拖至停机。这不是我们需要的那种容错系统。

这里的问题是，只有在所有的操作都被收集之后，操作的全序才会出现。如果另一个节点已经产生了一些操作，但你还不知道那些操作是什么，那就无法构造所有操作最终的全序关系：来自另一个节点的未知操作可能需要被插入到全序中的不同位置。

总之：为了实诸如如用户名上的唯一约束这种东西，仅有操作的全序是不够的，你还需要知道这个全序何时会尘埃落定。如果你有一个创建用户名的操作，并且确定在全序中，没有任何其他节点可以在你的操作之前插入对同一用户名的声称，那么你就可以安全地宣告操作执行成功。

如何知道你的全序关系已经尘埃落定，这个想法将在全序广播一节中详细说明。

全序广播

如果你的程序只运行在单个CPU核上，那么定义一个操作全序是很容易的：可以简单地就是CPU执行这些操作的顺序。但是在分布式系统中，让所有节点对同一个全局操作顺序达成一致可能相当棘手。在上一节中，我们讨论了按时间戳或序列号进行排序，但发现它还不如单主复制给力（如果你使用时间戳排序来实现唯一性约束，而且不能容忍任何错误）。

如前所述，单主复制通过选择一个节点作为主库来确定操作的全序，并在主库的单个CPU核上对所有操作进行排序。接下来的挑战是，如果吞吐量超出单个主库的处理能力，这种情况下如何扩展系统；以及，如果主库失效（“处理节点宕机”），如何处理故障转移。在分布式系统文献中，这个问题被称为全序广播（**total order broadcast**）或原子广播（**atomic broadcast**）[ix]【25,57,58】。

> ix. “原子广播”是一个传统的术语，非常混乱，而且与“原子”一词的其他用法不一致：它与ACID事务中的原子性没有任何关系，只是与原子操作（在多线程编程的意义上 ）或原子寄存器（线性一致存储）有间接的联系。全序广播是另一个同义词。↩

> 顺序保证的范围
>
> 每个分区各有一个主库的分区数据库，通常只在每个分区内维持顺序，这意味着它们不能提供跨分区的一致性保证（例如，一致性快照，外键引用）。 跨所有分区的全序是可能的，但需要额外的协调【59】。

全序广播通常被描述为在节点间交换消息的协议。 非正式地讲，它要满足两个安全属性：

可靠交付（***reliable delivery***）

没有消息丢失：如果消息被传递到一个节点，它将被传递到所有节点。

全序交付（**totally ordered delivery**）*

消息以相同的顺序传递给每个节点。

正确的全序广播算法必须始终保证可靠性和有序性，即使节点或网络出现故障。当然在网络中断的时候，消息是传不出去的，但是算法可以不断重试，以便在网络最终修复时，消息能及时通过并送达（当然它们必须仍然按照正确的顺序传递）。

使用全序广播

像ZooKeeper和etcd这样的共识服务实际上实现了全序广播。这一事实暗示了全序广播与共识之间有着紧密联系，我们将在本章稍后进行探讨。

全序广播正是数据库复制所需的：如果每个消息都代表一次数据库的写入，且每个副本都按相同的顺序处理相同的写入，那么副本间将相互保持一致（除了临时的复制延迟）。这个原理被称为状态机复制（**state machine replication**）【60】，我们将在第11章中重新回到这个概念。

与之类似，可以使用全序广播来实现可序列化的事务：如“真的串行执行”中所述，如果每个消息都表示一个确定性事务，以存储过程的形式来执行，且每个节点都以相同的顺序处理这些消息，那么数据库的分区和副本就可以相互保持一致【61】。

全序广播的一个重要表现是，顺序在消息送达时被固化：如果后续的消息已经送达，节点就不允许追溯地将（先前）消息插入顺序中的较早位置。这个事实使得全序广播比时间戳命令更强。

考量全序广播的另一种方式是，这是一种创建日志的方式（如在复制日志，事务日志或预写式日志中）：传递消息就像附加写入日志。由于所有节点必须以相同的顺序传递相同的消息，因此所有节点都可以读取日志，并看到相同的消息序列。

全序广播对于实现提供防护令牌的锁服务也很有用（参见“防护令牌”）。每个获取锁的请求都作为一条消息追加到日志末尾，并且所有的消息都按它们在日志中出现的顺序依次编号。序列号可以当成防护令牌用，因为它是单调递增的。在ZooKeeper中，这个序列号被称为 `zxid` 【15】。

使用全序广播实现线性一致的存储

如 图9-4 所示，在线性一致的系统中，存在操作的全序。这是否意味着线性一致与全序广播一样？不尽然，但两者之间有者密切的联系[x]。

x

> x. 从形式上讲，线性一致读写寄存器是一个“更容易”的问题。 全序广播等价于共识【67】，而共识问题在异步的崩溃-停止模型【68】中没有确定性的解决方案，而线性一致的读写寄存器可以在这种模型中实现【23,24,25】。 然而，支持诸如比较并设置（**CAS, compare-and-set**），或自增并返回（**increment-and-get**）的原子操作使它等价于共识问题【28】。 因此，共识问题与线性一致寄存器问题密切相关。 ↩

全序广播是异步的：消息被保证以固定的顺序可靠地传送，但是不能保证消息何时被送达（所以一个接收者可能落后于其他接收者）。相比之下，线性一致性是新鲜性的保证：读取一定能看见最新的写入值。

但如果有了全序广播，你就可以在此基础上构建线性一致的存储。例如，你可以确保用户名能唯一标识用户帐户。

设想对于每一个可能的用户名，你都可以有一个带有CAS原子操作的线性一致寄存器。每个寄存器最初的值为空值（表示不使用用户名）。当用户想要创建一个用户名时，对该用户名的寄存器执行CAS操作，在先前寄存器值为空的条件，将其值设置为用户的账号ID。如果多个用户试图同时获取相同的用户名，则只有一个CAS操作会成功，因为其他用户会看到非空的值（由于线性一致性）。

你可以通过将全序广播当成仅追加日志【62,63】的方式来实现这种线性一致的CAS操作：

1. 在日志中追加一条消息，试探性地指明你要声明的用户名。
2. 读日志，并等待你所附加的信息被回送。[xi]
3. 检查是否有任何消息声称目标用户名的所有权。如果这些消息中的第一条就你自己的消息，那么你就成功了：你可以提交声称的用户名（也许是通过向日志追加另一条消息）并向客户端确认。如果所需用户名的第一条消息来自其他用户，则中止操作。

> xi. 如果你不等待，而是在消息入队之后立即确认写入，则会得到类似于多核x86处理器内存的一致性模型【43】。 该模型既不是线性一致的也不是顺序一致的。 ↩

由于日志项是以相同顺序送达至所有节点，因此如果有多个并发写入，则所有节点会对最先到达者达成一致。选择冲突写入中的第一个作为胜利者，并中止后来者，以此确定所有节点对某个写入是提交还是中止达成一致。类似的方法可以在一个日志的基础上实现可序列化的多对象事务【62】。

尽管这一过程保证写入是线性一致的，但它并不保证读取也是线性一致的 —— 如果你从与日志异步更新的存储中读取数据，结果可能是陈旧的。 （精确地说，这里描述的过程提供了顺序一致性（**sequential consistency**）【47,64】，有时也称为时间线一致性（**timeline consistency**）【65,66】，比线性一致性稍微弱一些的保证）。为了使读取也线性一致，有几个选项：

- 你可以通过追加一条消息，当消息回送时读取日志，执行实际的读取。消息在日志中的位置因此定义了读取发生的时间点。 （etcd的法定人数读取有些类似这种情况【16】。）

- 如果日志允许以线性一致的方式获取最新日志消息的位置，则可以查询该位置，等待直到该位置前的所有消息都传达到你，然后执行读取。（这是Zookeeper `sync()` 操作背后的思想【15】）。
- 你可以从同步更新的副本中进行读取，因此可以确保结果是最新的。（这种技术用于链式复制【63】；参阅“复制研究”。）

使用线性一致性存储实现全序广播

上一节介绍了如何从全序广播构建一个线性一致的CAS操作。我们也可以把它反过来，假设我们有线性一致的存储，接下来会展示如何在此基础上构建全序广播。

最简单的方法是假设你有一个线性一致的寄存器来存储一个整数，并且有一个原子自增并返回操作【28】。或者原子CAS操作也可以完成这项工作。

该算法很简单：每个要通过全序广播发送的消息首先对线性一致寄存器执行自增并返回操作。然后将从寄存器获得的值作为序列号附加到消息中。然后你可以将消息发送到所有节点（重新发送任何丢失的消息），而收件人将按序列号连续发送消息。

请注意，与兰伯特时间戳不同，通过自增线性一致性寄存器获得的数字形式上是一个没有间隙的序列。因此，如果一个节点已经发送了消息 4 并且接收到序列号为 6 的传入消息，则它知道它在传递消息 6 之前必须等待消息 5 。兰伯特时间戳则与之不同 —— 事实上，这是全序广播和时间戳排序间的关键区别。

实现一个带有原子性自增并返回操作的线性一致寄存器有多困难？像往常一样，如果事情从来不出差错，那很容易：你可以简单地把它保存在单个节点内的变量中。问题在于处理当该节点的网络连接中断时的情况，并在该节点失效时能恢复这个值【59】。一般来说，如果你对线性一致性的序列号生成器进行深入过足够深入的思考，你不可避免地会得出一个共识算法。

这并非巧合：可以证明，线性一致的CAS（或自增并返回）寄存器与全序广播都都等价于共识问题【28,67】。也就是说，如果你能解决其中的一个问题，你可以把它转化成为其他问题的解决方案。这是相当深刻和令人惊讶的洞察！

现在是时候正面处理共识问题了，我们将在本章的其余部分进行讨论。

分布式事务与共识

共识是分布式计算中最重要也是最基本的问题之一。从表面上看似乎很简单：非正式地讲，目标只是让几个节点达成一致（**get serveral nodes to agree on something**）。你也许会认为这不会太难。不幸的是，许多出故障的系统都是因为错误地轻信这个问题很容易解决。

尽管共识非常重要，但关于它的内容出现在本书的后半部分，因为这个主题非常微妙，欣赏细微之处需要一些必要的知识。即使在学术界，对共识的理解也是在几十年的过程中逐渐沉淀而来，一路上也有着许多误解。现在我们已经讨论了复制（第5章），事务（第7章），系

统模型（第8章），线性一致以及全序（本章），我们终于准备好解决共识问题了。

节点能达成一致，在很多场景下都非常重要，例如：

领导选举

在单主复制的数据库中，所有节点需要就哪个节点是领导者达成一致。如果一些节点由于网络故障而无法与其他节点通信，则可能会对领导权的归属引起争议。在这种情况下，共识对于避免错误的故障切换非常重要。错误的故障切换会导致两个节点都认为自己是领导者（脑裂，参阅“处理节点宕机”）。如果有两个领导者，它们都会接受写入，它们的数据会发生分歧，从而导致不一致和数据丢失。

原子提交

在支持跨多节点或跨多分区事务的数据库中，一个事务可能在某些节点上失败，但在其他节点上成功。如果我们想要维护事务的原子性（就ACID而言，请参“原子性”），我们必须让所有节点对事务的结果达成一致：要么全部中止/回滚（如果出现任何错误），要么它们全部提交（如果没有出错）。这个共识的例子被称为原子提交（**atomic commit**）问题[xii]。

> [xii]. 原子提交的形式化与共识稍有不同：原子事务只有在所有参与者投票提交的情况下才能提交，如果有任何参与者需要中止，则必须中止。 共识则允许就任意一个被参与者提出的候选值达成一致。 然而，原子提交和共识可以相互简化为对方【70,71】。 非阻塞原子提交则要比共识更为困难 —— 参阅“三阶段提交”。 ↩

> ## 共识的不可能性
>
> 你可能已经听说过作者Fischer，Lynch和Paterson之后的FLP结果【68】，它证明，如果存在节点可能崩溃的风险，则不存在总是能够达成共识的算法。在分布式系统中，我们必须假设节点可能会崩溃，所以可靠的共识是不可能的。然而这里我们正在讨论达成共识的算法，到底是怎么回事？
>
> 答案是FLP结果在异步系统模型中得到了证明（参阅“系统模型与现实”），这是一种限制性很强的模型，它假定确定性算法不能使用任何时钟或超时。如果允许算法使用超时或其他方法来识别可疑的崩溃节点（即使怀疑有时是错误的），则共识变为一个可解的问题【67】。即使仅仅允许算法使用随机数，也足以绕过这个不可能的结果【69】。
>
> 因此，FLP是关于共识不可能性的重要理论结果，但现实中的分布式系统通常是可以达成共识的。

在本节中，我们将首先更详细地研究原子提交问题。具体来说，我们将讨论两阶段提交（**2PC, two-phase commit**）算法，这是解决原子提交问题最常见的办法，并在各种数据库、消息队列和应用服务器中实现。事实证明2PC是一种共识算法，但不是一个非常好的算法【70,71】。

通过对2PC的学习，我们将继续努力实现更好的一致性算法，比如ZooKeeper（Zab）和etcd（Raft）中使用的算法。

原子提交与二阶段提交（2PC）

在第7章中我们了解到，事务原子性的目的是在多次写操作中途出错的情况下，提供一种简单的语义。事务的结果要么是成功提交，在这种情况下，事务的所有写入都是持久化的；要么是中止，在这种情况下，事务的所有写入都被回滚（即撤消或丢弃）。

原子性可以防止失败的事务搅乱数据库，避免数据库陷入半成品结果和半更新状态。这对于多对象事务（参阅“单对象和多对象操作”）和维护次级索引的数据库尤其重要。每个辅助索引都是与主数据相分离的数据结构—— 因此，如果你修改了一些数据，则还需要在辅助索引中进行相应的更改。原子性确保二级索引与主数据保持一致（如果索引与主数据不一致，就没什么用了）。

从单节点到分布式原子提交

对于在单个数据库节点执行的事务，原子性通常由存储引擎实现。当客户端请求数据库节点提交事务时，数据库将使事务的写入持久化（通常在预写式日志中：参阅“使B树可靠”），然后将提交记录追加到磁盘中的日志里。如果数据库在这个过程中间崩溃，当节点重启时，事务会从日志中恢复：如果提交记录在崩溃之前成功地写入磁盘，则认为事务被提交；否则来自该事务的任何写入都被回滚。

因此，在单个节点上，事务的提交主要取决于数据持久化落盘的顺序：首先是数据，然后是提交记录【72】。事务提交或终止的关键决定时刻是磁盘完成写入提交记录的时刻：在此之前，仍有可能中止（由于崩溃），但在此之后，事务已经提交（即使数据库崩溃）。因此，是单一的设备（连接到单个磁盘驱动的控制器，且挂载在单台机器上）使得提交具有原子性。

但是，如果一个事务中涉及多个节点呢？例如，你也许在分区数据库中会有一个多对象事务，或者是一个按关键词分区的二级索引（其中索引条目可能位于与主数据不同的节点上；参阅“分区和二级索引”）。大多数“NoSQL”分布式数据存储不支持这种分布式事务，但是很多关系型数据库集群支持（参见“实践中的分布式事务”）。

在这些情况下，仅向所有节点发送提交请求并独立提交每个节点的事务是不够的。这样很容易发生违反原子性的情况：提交在某些节点上成功，而在其他节点上失败：

- 某些节点可能会检测到约束冲突或冲突，因此需要中止，而其他节点则可以成功进行提交。
- 某些提交请求可能在网络中丢失，最终由于超时而中止，而其他提交请求则通过。
- 在提交记录完全写入之前，某些节点可能会崩溃，并在恢复时回滚，而其他节点则成功提交。

如果某些节点提交了事务，但其他节点却放弃了这些事务，那么这些节点就会彼此不一致（如 图7-3 所示）。而且一旦在某个节点上提交了一个事务，如果事后发现它在其它节点上被中止了，它是无法撤回的。出于这个原因，一旦确定事务中的所有其他节点也将提交，节点就必须进行提交。

事务提交必须是不可撤销的 —— 事务提交之后，你不能改变主意，并追溯性地中止事务。这个规则的原因是，一旦数据被提交，其结果就对其他事务可见，因此其他客户端可能会开始依赖这些数据。这个原则构成了读已提交隔离等级的基础，在“读已提交”一节中讨论了这个问题。如果一个事务在提交后被允许中止，所有那些读取了已提交却又被追溯声明不存在数据的事务也必须回滚。

（提交事务的结果有可能通过事后执行另一个补偿事务来取消【73,74】，但从数据库的角度来看，这是一个单独的事务，因此任何关于跨事务正确性的保证都是应用自己的问题。）

两阶段提交简介

两阶段提交（**two-phase commit**）是一种用于实现跨多个节点的原子事务提交的算法，即确保所有节点提交或所有节点中止。 它是分布式数据库中的经典算法【13,35,75】。 2PC在某些数据库内部使用，也以**XA**事务的形式对应用可用【76,77】（例如Java Transaction API支持）或以SOAP Web服务的 `WS-AtomicTransaction` 形式提供给应用程序【78,79】。

图9-9说明了2PC的基本流程。2PC中的提交/中止过程分为两个阶段（因此而得名），而不是单节点事务中的单个提交请求。

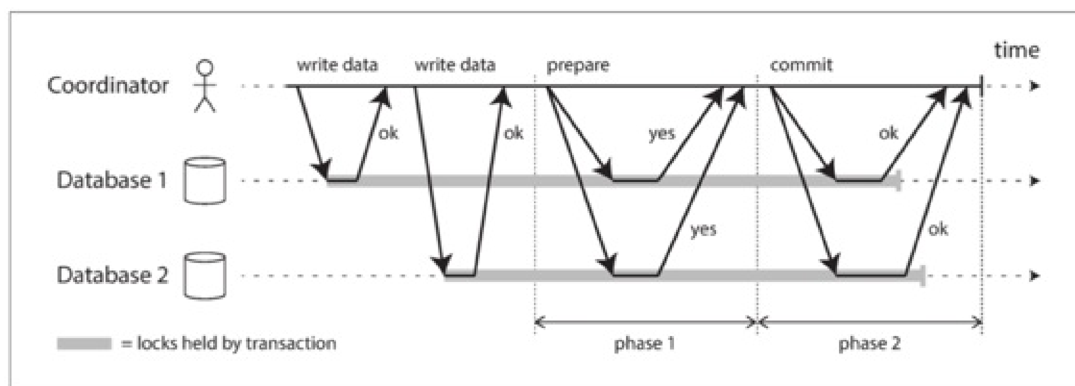


图**9-9** 两阶段提交（**2PC**）的成功执行

> ### 不要把2PC和2PL搞混了
>
> 两阶段提交（2PC）和两阶段锁定（参阅“两阶段锁定（2PL）”）是两个完全不同的东西。 2PC在分布式数据库中提供原子提交，而2PL提供可序列化的隔离等级。为了避免混淆，最好把它们看作完全独立的概念，并忽略名称中不幸的相似性。

2PC使用一个通常不会出现在单节点事务中的新组件：协调者（**coordinator**）（也称为事务管理器（**transaction manager**））。协调者通常在请求事务的相同应用程序进程中以库的形式实现（例如，嵌入在Java EE容器中），但也可以是单独的进程或服务。这种协调者的例子

包括Narayana，JOTM，BTM或MSDTC。

正常情况下，2PC事务以应用程序在多个数据库节点上读写数据开始。我们称这些数据库节点为参与者（**participants**）。当应用程序准备提交时，协调者开始阶段 1 ：它发送一个准备（**prepare**）请求到每个节点，询问它们是否能够提交。然后协调者会跟踪参与者的响应：

- 如果所有参与者都回答“是”，表示它们已经准备好提交，那么协调者在阶段 2 发出提交（**commit**）请求，然后提交真正发生。
- 如果任意一个参与者回复了“否”，则协调者在阶段2 中向所有节点发送中止（**abort**）请求。

这个过程有点像西方传统婚姻仪式：司仪分别询问新娘和新郎是否要结婚，通常是从两方都收到“我愿意”的答复。收到两者的回复后，司仪宣布这对情侣成为夫妻：事务就提交了，这一幸福事实会广播至所有的参与者中。如果新娘与新郎之一没有回复”我愿意“，婚礼就会中止【73】。

系统承诺

这个简短的描述可能并没有说清楚为什么两阶段提交保证了原子性，而跨多个节点的一阶段提交却没有。在两阶段提交的情况下，准备请求和提交请求当然也可以轻易丢失。 2PC又有什么不同呢？

为了理解它的工作原理，我们必须更详细地分解这个过程：

1. 当应用想要启动一个分布式事务时，它向协调者请求一个事务ID。此事务ID是全局唯一的。
2. 应用在每个参与者上启动单节点事务，并在单节点事务上捎带上这个全局事务ID。所有的读写都是在这些单节点事务中各自完成的。如果在这个阶段出现任何问题（例如，节点崩溃或请求超时），则协调者或任何参与者都可以中止。
3. 当应用准备提交时，协调者向所有参与者发送一个准备请求，并打上全局事务ID的标记。如果任意一个请求失败或超时，则协调者向所有参与者发送针对该事务ID的中止请求。
4. 参与者收到准备请求时，需要确保在任意情况下都的确可以提交事务。这包括将所有事务数据写入磁盘（出现故障，电源故障，或硬盘空间不足都不能是稍后拒绝提交的理由）以及检查是否存在任何冲突或违反约束。通过向协调者回答“是”，节点承诺，只要请求，这个事务一定可以不出差错地提交。换句话说，参与者放弃了中止事务的权利，但没有实际提交。
5. 当协调者收到所有准备请求的答复时，会就提交或中止事务作出明确的决定（只有在所有参与者投赞成票的情况下才会提交）。协调者必须把这个决定写到磁盘上的事务日志中，如果它随后就崩溃，恢复后也能知道自己所做的决定。这被称为提交点（**commit point**）。
6. 一旦协调者的决定落盘，提交或放弃请求会发送给所有参与者。如果这个请求失败或超时，协调者必须永远保持重试，直到成功为止。没有回头路：如果已经做出决定，不管

需要多少次重试它都必须被执行。如果参与者在此期间崩溃，事务将在其恢复后提交——由于参与者投了赞成，因此恢复后它不能拒绝提交。

因此，该协议包含两个关键的“不归路”点：当参与者投票“是”时，它承诺它稍后肯定能够提交（尽管协调者可能仍然选择放弃）。一旦协调者做出决定，这一决定是不可撤销的。这些承诺保证了2PC的原子性。（单节点原子提交将这两个事件混为一谈：将提交记录写入事务日志。）

回到婚姻的比喻，在说“我是”之前，你和你的新娘/新郎有中止这个事务的自由，通过回复“没门！”（或者有类似效果的话）。然而在说了“我愿意”之后，你就不能撤回那个声明了。如果你说“我愿意”后晕倒了，没有听到司仪说“你们现在是夫妻了”，那也并不会改变事务已经提交的现实。当你稍后恢复意识时，可以通过查询司仪的全局事务ID状态来确定你是否已经成婚，或者你可以等待司仪重试下一次提交请求（因为重试将在你无意识期间一直持续）。

协调者失效

我们已经讨论了在2PC期间，如果参与者之一或网络发生故障时会发生什么情况：如果任何一个准备请求失败或者超时，协调者就会中止事务。如果任何提交或中止请求失败，协调者将无条件重试。但是如果协调者崩溃，会发生什么情况就不太清楚了。

如果协调者在发送准备请求之前失败，参与者可以安全地中止事务。但是，一旦参与者收到了准备请求并投了“是”，就不能再单方面放弃 —— 必须等待协调者回答事务是否已经提交或中止。如果此时协调者崩溃或网络出现故障，参与者什么也做不了只能等待。参与者的这种事务状态称为存疑（**in doubt**）的或不确定（**uncertain**）的。

情况如图9-10 所示。在这个特定的例子中，协调者实际上决定提交，数据库2 收到提交请求。但是，协调者在将提交请求发送到数据库1 之前发生崩溃，因此数据库1 不知道是否提交或中止。即使超时在这里也没有帮助：如果数据库1 在超时后单方面中止，它将最终与执行提交的数据库2 不一致。同样，单方面提交也是不安全的，因为另一个参与者可能已经中止了。

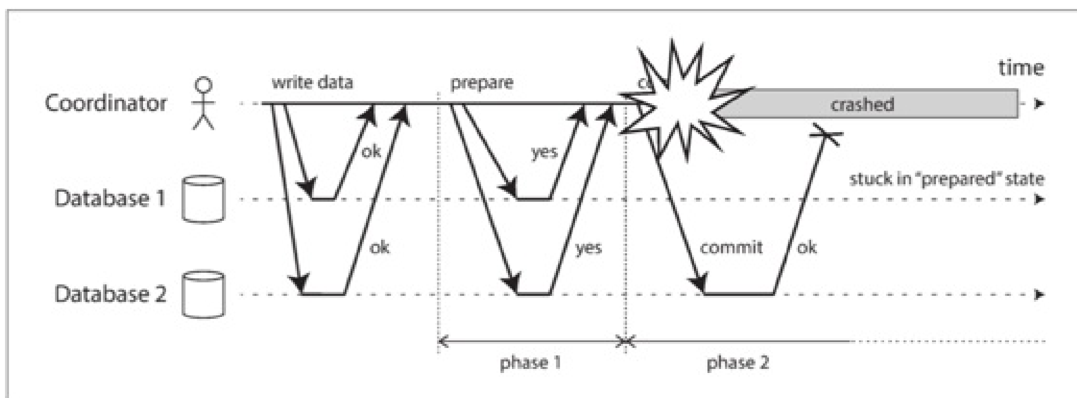


图**9-10** 参与者投赞成票后，协调者崩溃。数据库**1**不知道是否提交或中止

没有协调者的消息，参与者无法知道是提交还是放弃。原则上参与者可以相互沟通，找出每个参与者是如何投票的，并达成一致，但这不是2PC协议的一部分。

可以完成2PC的唯一方法是等待协调者恢复。这就是为什么协调者必须在向参与者发送提交或中止请求之前，将其提交或中止决定写入磁盘上的事务日志：协调者恢复后，通过读取其事务日志来确定所有存疑事务的状态。任何在协调者日志中没有提交记录的事务都会中止。因此，2PC的提交点归结为协调者上的常规单节点原子提交。

三阶段提交

两阶段提交被称为阻塞（**blocking**）原子提交协议，因为存在2PC可能卡住并等待协调者恢复的情况。理论上，可以使一个原子提交协议变为非阻塞（**nonblocking**）的，以便在节点失败时不会卡住。但是让这个协议能在实践中工作并没有那么简单。

作为2PC的替代方案，已经提出了一种称为三阶段提交（**3PC**）的算法【13,80】。然而，3PC假定网络延迟有界，节点响应时间有限；在大多数具有无限网络延迟和进程暂停的实际系统中（见第8章），它并不能保证原子性。

通常，非阻塞原子提交需要一个完美的故障检测器（**perfect failure detector**）【67,71】—— 即一个可靠的机制来判断一个节点是否已经崩溃。在具有无限延迟的网络中，超时并不是一种可靠的故障检测机制，因为即使没有节点崩溃，请求也可能由于网络问题而超时。出于这个原因，2PC仍然被使用，尽管大家都清楚可能存在协调者故障的问题。

实践中的分布式事务

分布式事务的名声毁誉参半，尤其是那些通过两阶段提交实现的。一方面，它被视作提供了一个难以实现的重要的安全性保证；另一方面，它们因为导致运维问题，造成性能下降，做出超过能力范围的承诺而饱受批评【81,82,83,84】。许多云服务由于其导致的运维问题，而选择不实现分布式事务【85,86】。

分布式事务的某些实现会带来严重的性能损失 —— 例如据报告称，MySQL中的分布式事务比单节点事务慢10倍以上【87】，所以当人们建议不要使用它们时就不足为奇了。两阶段提交所固有的性能成本，大部分是由于崩溃恢复所需的额外强制刷盘（`fsync`）【88】以及额外的网络往返。

但我们不应该直接忽视分布式事务，而应当更加仔细地审视这些事务，因为从中可以汲取重要的经验教训。首先，我们应该精确地说明“分布式事务”的含义。两种截然不同的分布式事务类型经常被混淆：

数据库内部的分布式事务

一些分布式数据库（即在其标准配置中使用复制和分区的数据库）支持数据库节点之间的内部事务。例如，VoltDB和MySQL Cluster的NDB存储引擎就有这样的内部事务支持。在这种情况下，所有参与事务的节点都运行相同的数据库软件。

异构分布式事务

在异构（**heterogeneous**）事务中，参与者是两种或以上不同技术：例如来自不同供应商的两个数据库，甚至是非数据库系统（如消息代理）。跨系统的分布式事务必须确保原子提交，尽管系统可能完全不同。

数据库内部事务不必与任何其他系统兼容，因此它们可以使用任何协议，并能针对特定技术进行特定的优化。因此数据库内部的分布式事务通常工作地很好。另一方面，跨异构技术的事务则更有挑战性。

恰好一次的消息处理

异构的分布式事务处理能够以强大的方式集成不同的系统。例如：消息队列中的一条消息可以被确认为已处理，当且仅当用于处理消息的数据库事务成功提交。这是通过在同一个事务中原子提交消息确认和数据库写入两个操作来实现的。藉由分布式事务的支持，即使消息代理和数据库是在不同机器上运行的两种不相关的技术，这种操作也是可能的。

如果消息传递或数据库事务任意一者失败，两者都会中止，因此消息代理可能会在稍后安全地重传消息。因此，通过原子提交消息处理及其副作用，即使在成功之前需要几次重试，也可以确保消息被有效地（**effectively**）恰好处理一次。中止会抛弃部分完成事务所导致的任何副作用。

然而，只有当所有受事务影响的系统都使用同样的原子提交协议（**atomic commit protocl**）时，这样的分布式事务才是可能的。例如，假设处理消息的副作用是发送一封邮件，而邮件服务器并不支持两阶段提交：如果消息处理失败并重试，则可能会发送两次或更多次的邮件。但如果处理消息的所有副作用都可以在事务中止时回滚，那么这样的处理流程就可以安全地重试，就好像什么都没有发生过一样。

在第11章中将再次回到”恰好一次“消息处理的主题。让我们先来看看允许这种异构分布式事务的原子提交协议。

XA事务

X/Open XA（扩展架构（**eXtended Architecture**）的缩写）是跨异构技术实现两阶段提交的标准【76,77】。它于1991年推出并得到了广泛的实现：许多传统关系数据库（包括PostgreSQL，MySQL，DB2，SQL Server和Oracle）和消息代理（包括ActiveMQ，HornetQ，MSMQ和IBM MQ） 都支持XA。

XA不是一个网络协议——它只是一个用来与事务协调者连接的C API。其他语言也有这种API的绑定；例如在Java EE应用的世界中，XA事务是使用**Java**事务**API**（**JTA, Java Transaction API**）实现的，而许多使用**Java**数据库连接（**JDBC, Java Database Connectivity**）的数据库驱动，以及许多使用**Java**消息服务（**JMS**）API的消息代理都支持**Java**事务**API**（**JTA**）。

XA假定你的应用使用网络驱动或客户端库来与参与者进行通信（数据库或消息服务）。如果驱动支持XA，则意味着它会调用XA API 以查明操作是否为分布式事务的一部分 —— 如果是，则将必要的信息发往数据库服务器。驱动还会向协调者暴露回调接口，协调者可以通过回调来要求参与者准备，提交或中止。

事务协调者需要实现XA API。标准没有指明应该如何实现，但实际上协调者通常只是一个库，被加载到发起事务的应用的同一个进程中（而不是单独的服务）。它在事务中个跟踪所有的参与者，并在要求它们准备之后收集参与者的响应（通过驱动回调），并使用本地磁盘上的日志记录每次事务的决定（提交/中止）。

如果应用进程崩溃，或者运行应用的机器报销了，协调者也随之往生极乐。然后任何带有准备了但未提交事务的参与者都会在疑虑中卡死。由于协调程序的日志位于应用服务器的本地磁盘上，因此必须重启该服务器，且协调程序库必须读取日志以恢复每个事务的提交/中止结果。只有这样，协调者才能使用数据库驱动的XA回调来要求参与者提交或中止。数据库服务器不能直接联系协调者，因为所有通信都必须通过客户端库。

怀疑时持有锁

为什么我们这么关心存疑事务？系统的其他部分就不能继续正常工作，无视那些终将被清理的存疑事务吗？

问题在于锁（**locking**）。正如在“读已提交”中所讨论的那样，数据库事务通常获取待修改的行上的行级排他锁，以防止脏写。此外，如果要使用可序列化的隔离等级，则使用两阶段锁定的数据库也必须为事务所读取的行加上共享锁（参见“两阶段锁定（2PL）”）。

在事务提交或中止之前，数据库不能释放这些锁（如图9-9中的阴影区域所示）。因此，在使用两阶段提交时，事务必须在整个存疑期间持有这些锁。如果协调者已经崩溃，需要20分钟才能重启，那么这些锁将会被持有20分钟。如果协调者的日志由于某种原因彻底丢失，这些锁将被永久持有 —— 或至少在管理员手动解决该情况之前。

当这些锁被持有时，其他事务不能修改这些行。根据数据库的不同，其他事务甚至可能因为读取这些行而被阻塞。因此，其他事务没法儿简单地继续它们的业务了 —— 如果它们要访问同样的数据，就会被阻塞。这可能会导致应用大面积进入不可用状态，直到存疑事务被解决。

从协调者故障中恢复

理论上，如果协调者崩溃并重新启动，它应该干净地从日志中恢复其状态，并解决任何存疑事务。然而在实践中，孤立（**orphaned**）的存疑事务确实会出现【89,90】，即无论出于何种理由，协调者无法确定事务的结果（例如事务日志已经由于软件错误丢失或损坏）。这些事务无法自动解决，所以它们永远待在数据库中，持有锁并阻塞其他事务。

即使重启数据库服务器也无法解决这个问题，因为在2PC的正确实现中，即使重启也必须保留存疑事务的锁（否则就会冒有违反原子性保证的风险）。这是一种棘手的情况。

唯一的出路是让管理员手动决定提交还是回滚事务。管理员必须检查每个存疑事务的参与者，确定是否有任何参与者已经提交或中止，然后将相同的结果应用于其他参与者。解决这个问题潜在地需要大量的人力，并且可能发生在严重的生产中断期间（不然为什么协调者处于这种糟糕的状态），并很可能要在巨大精神压力和时间压力下完成。

许多XA的实现都有一个叫做启发式决策（**heuristic decistions**）的紧急逃生舱口：允许参与者单方面决定放弃或提交一个存疑事务，而无需协调者做出最终决定【76,77,91】。要清楚的是，这里启发式是可能破坏原子性（**probably breaking atomicity**）的委婉说法，因为它违背了两阶段提交的系统承诺。因此，启发式决策只是为了逃出灾难性的情况而准备的，而不是为了日常使用的。

分布式事务的限制

XA事务解决了保持多个参与者（数据系统）相互一致的现实的重要问题，但正如我们所看到的那样，它也引入了严重的运维问题。特别来讲，这里的核心认识是：事务协调者本身就是一种数据库（存储了事务的结果），因此需要像其他重要数据库一样小心地打交道：

- 如果协调者没有复制，而是只在单台机器上运行，那么它是整个系统的失效单点（因为它的失效会导致其他应用服务器阻塞在存疑事务持有的锁上）。令人惊讶的是，许多协调者实现默认情况下并不是高可用的，或者只有基本的复制支持。
- 许多服务器端应用都是使用无状态模式开发的（受HTTP的青睐），所有持久状态都存储在数据库中，因此具有应用服务器可随意按需添加删除的优点。但是，当协调者成为应用服务器的一部分时，它会改变部署的性质。突然间，协调者的日志成为持久系统状态的关键部分—— 与数据库本身一样重要，因为协调者日志是为了在崩溃后恢复存疑事务所必需的。这样的应用服务器不再是无状态的了。
- 由于XA需要兼容各种数据系统，因此它必须是所有系统的最小公分母。例如，它不能检测不同系统间的死锁（因为这将需要一个标准协议来让系统交换每个事务正在等待的锁的信息），而且它无法与SSI 协同工作，因为这需要一个跨系统定位冲突的协议。
- 对于数据库内部的分布式事务（不是XA），限制没有这么大，例如，分布式版本的SSI是可能的。然而仍然存在问题：2PC成功提交一个事务需要所有参与者的响应。因此，如果系统的任何部分损坏，事务也会失败。因此，分布式事务又有扩大失效（**amplifying failures**）的趋势，这又与我们构建容错系统的目标背道而驰。

这些事实是否意味着我们应该放弃保持几个系统相互一致的所有希望？不完全是 —— 还有其他的办法，可以让我们在没有异构分布式事务的痛苦的情况下实现同样的事情。我们将在第11章 和第12章 回到这些章节。但首先，我们应该概括一下关于共识的话题。

容错共识

非正式地，共识意味着让几个节点就某事达成一致。例如，如果有几个人同时（**concurrently**）尝试预订飞机上的最后一个座位，或剧院中的同一个座位，或者尝试使用相同的用户名注册一个帐户。共识算法可以用来确定这些互不相容（**mutually**

incompatible）的操作中，哪一个才是赢家。

共识问题通常形式化如下：一个或多个节点可以提议（**propose**）某些值，而共识算法决定（**decides**）采用其中的某个值。在座位预订的例子中，当几个顾客同时试图订购最后一个座位时，处理顾客请求的每个节点可以提议正在服务的顾客的ID，而决定指明了哪个顾客获得了座位。

在这种形式下，共识算法必须满足以下性质【25】：[xiii]

> [xiii]. 这种共识的特殊形式被称为统一共识（**uniform consensus**），相当于在具有不可靠故障检测器的异步系统中的常规共识（**regular consensus**）【71】。学术文献通常指的是进程（**process**）而不是节点，但我们在这里使用节点（**node**）来与本书的其余部分保持一致。↩

一致同意（***Uniform agreement***）

没有两个节点的决定不同。

完整性（***Integrity***）

没有节点决定两次。

有效性（***Validity***）

如果一个节点决定了值 `v` ，则 `v` 由某个节点所提议。

终止（***Termination***） 由所有未崩溃的节点来最终决定值。

一致同意和完整性属性定义了共识的核心思想：所有人都决定了相同的结果，一旦决定了，你就不能改变主意。有效性属性主要是为了排除平凡的解决方案：例如，无论提议了什么值，你都可以有一个始终决定值为 `null` 的算法。；该算法满足一致同意和完整性属性，但不满足有效性属性。

如果你不关心容错，那么满足前三个属性很容易：你可以将一个节点硬编码为“独裁者”，并让该节点做出所有的决定。但如果该节点失效，那么系统就无法再做出任何决定。事实上，这就是我们在两阶段提交的情况中所看到的：如果协调者失效，那么存疑的参与者就无法决定提交还是中止。

终止属性正式形成了容错的思想。它实质上说的是，一个共识算法不能简单地永远闲坐着等死 —— 换句话说，它必须取得进展。即使部分节点出现故障，其他节点也必须达成一项决定。（终止是一种活性属性，而另外三种是安全属性 —— 参见“安全性和活性”。）

共识的系统模型假设，当一个节点“崩溃”时，它会突然消失而且永远不会回来。（不像软件崩溃，想象一下地震，包含你的节点的数据中心被山体滑坡所摧毁，你必须假设节点被埋在30英尺以下的泥土中，并且永远不会重新上线）在这个系统模型中，任何需要等待节点恢复的算法都不能满足终止属性。特别是，2PC不符合终止属性的要求。

当然如果所有的节点都崩溃了，没有一个在运行，那么所有算法都不可能决定任何事情。算法可以容忍的失效数量是有限的：事实上可以证明，任何共识算法都需要至少占总体多数（**majority**）的节点正确工作，以确保终止属性【67】。多数可以安全地组成法定人数（参阅“读写的法定人数”）。

因此终止属性取决于一个假设，不超过一半的节点崩溃或不可达。然而即使多数节点出现故障或存在严重的网络问题，绝大多数共识的实现都能始终确保安全属性得到满足—— 一致同意，完整性和有效性【92】。因此，大规模的中断可能会阻止系统处理请求，但是它不能通过使系统做出无效的决定来破坏共识系统。

大多数共识算法假设不存在拜占庭式错误，正如在“拜占庭式错误”一节中所讨论的那样。也就是说，如果一个节点没有正确地遵循协议（例如，如果它向不同节点发送矛盾的消息），它就可能会破坏协议的安全属性。克服拜占庭故障，稳健地达成共识是可能的，只要少于三分之一的节点存在拜占庭故障【25,93】。但我们没有地方在本书中详细讨论这些算法了。

共识算法和全序广播

最著名的容错共识算法是视图戳复制（**VSR, viewstamped replication**）【94,95】，Paxos【96,97,98,99】，Raft【22,100,101】以及 Zab【15,21,102】。这些算法之间有不少相似之处，但它们并不相同【103】。在本书中我们不会介绍各种算法的详细细节：了解一些它们共通的高级思想通常已经足够了，除非你准备自己实现一个共识系统。（可能并不明智，相当难【98,104】）

大多数这些算法实际上并不直接使用这里描述的形式化模型（提议与决定单个值，一致同意，完整性，有效性和终止属性）。取而代之的是，它们决定了值的顺序（**sequence**），这使它们成为全序广播算法，正如本章前面所讨论的那样（参阅“全序广播”）。

请记住，全序广播要求将消息按照相同的顺序，恰好传递一次，准确传送到所有节点。如果仔细思考，这相当于进行了几轮共识：在每一轮中，节点提议下一条要发送的消息，然后决定在全序中下一条要发送的消息【67】。

所以，全序广播相当于重复进行多轮共识（每次共识决定与一次消息传递相对应）：

- 由于一致同意属性，所有节点决定以相同的顺序传递相同的消息。
- 由于完整性属性，消息不会重复。
- 由于有效性属性，消息不会被损坏，也不能凭空编造。
- 由于终止属性，消息不会丢失。

视图戳复制，Raft和Zab直接实现了全序广播，因为这样做比重复一次一值（**one value a time**）的共识更高效。在Paxos的情况下，这种优化被称为Multi-Paxos。

单领导者复制和共识

在第5章中，我们讨论了单领导者复制（参见“领导者和追随者”），它将所有的写入操作都交给主库，并以相同的顺序将它们应用到从库，从而使副本保持在最新状态。这实际上不就是一个全序广播吗？为什么我们在第五章里一点都没担心过共识问题呢？

答案取决于如何选择领导者。如果主库是由运维人员手动选择和配置的，那么你实际上拥有一种独裁类型的“共识算法”：只有一个节点被允许接受写入（即决定写入复制日志的顺序），如果该节点发生故障，则系统将无法写入，直到运维手动配置其他节点作为主库。这样的系统在实践中可以表现良好，但它无法满足共识的终止属性，因为它需要人为干预才能取得进展。

一些数据库会自动执行领导者选举和故障转移，如果旧主库失效，会提拔一个从库为新主库（参见“处理节点宕机”）。这使我们向容错的全序广播更进一步，从而达成共识。

但是还有一个问题。我们之前曾经讨论过脑裂的问题，并且说过所有的节点都需要同意是谁领导，否则两个不同的节点都会认为自己是领导者，从而导致数据库进入不一致的状态。因此，选出一位领导者需要共识。但如果这里描述的共识算法实际上是全序广播算法，并且全序广播就像单主复制，而单主复制需要一个领导者，那么...

这样看来，要选出一个领导者，我们首先需要一个领导者。要解决共识问题，我们首先需要解决共识问题。我们如何跳出这个先有鸡还是先有蛋的问题？

时代编号和法定人数

迄今为止所讨论的所有共识协议，在内部都以某种形式使用一个领导者，但它们并不能保证领导者是独一无二的。相反，它们可以做出更弱的保证：协议定义了一个时代编号（**epoch number**）（在Paxos中称为投票编号（**ballot number**），视图戳复制中的视图编号（**view number**），以及Raft中的任期号码（**term number**）），并确保在每个时代中，领导者都是唯一的。

每次当现任领导被认为挂掉的时候，节点间就会开始一场投票，以选出一个新领导。这次选举被赋予一个递增的时代编号，因此时代编号是全序且单调递增的。如果两个不同的时代的领导者之间出现冲突（也许是因为前任领导者实际上并未死亡），那么带有更高时代编号的领导说了算。

在任何领导者被允许决定任何事情之前，必须先检查是否存在其他带有更高时代编号的领导者，它们可能会做出相互冲突的决定。领导者如何知道自己没有被另一个节点赶下台？回想一下在“真理在多数人手中”中提到的：一个节点不一定能相信自己的判断—— 因为只有节点自己认为自己是领导者，并不一定意味着其他节点接受它作为它们的领导者。

相反，它必须从法定人数（**quorum**）的节点中获取选票（参阅“读写的法定人数”）。对领导者想要做出的每一个决定，都必须将提议值发送给其他节点，并等待法定人数的节点响应并赞成提案。法定人数通常（但不总是）由多数节点组成【105】。只有在没有意识到任何带有更高时代编号的领导者的情况下，一个节点才会投票赞成提议。

因此，我们有两轮投票：第一次是为了选出一位领导者，第二次是对领导者的提议进行表决。关键的洞察在于，这两次投票的法定人群必须相互重叠（**overlap**）：如果一个提案的表决通过，则至少得有一个参与投票的节点也必须参加过最近的领导者选举【105】。因此，如果在一个提案的表决过程中没有出现更高的时代编号。那么现任领导者就可以得出这样的结论：没有发生过更高时代的领导选举，因此可以确定自己仍然在领导。然后它就可以安全地对提议值做出决定。

这一投票过程表面上看起来很像两阶段提交。最大的区别在于，2PC中协调者不是由选举产生的，而且2PC则要求所有参与者都投赞成票，而容错共识算法只需要多数节点的投票。而且，共识算法还定义了一个恢复过程，节点可以在选举出新的领导者之后进入一个一致的状态，确保始终能满足安全属性。这些区别正是共识算法正确性和容错性的关键。

共识的局限性

共识算法对于分布式系统来说是一个巨大的突破：它为其他充满不确定性的系统带来了基础的安全属性（一致同意，完整性和有效性），然而它们还能保持容错（只要多数节点正常工作且可达，就能取得进展）。它们提供了全序广播，因此也可以它们也可以以一种容错的方式实现线性一致的原子操作（参见“使用全序广播实现线性一致性存储”）。

尽管如此，它们并不是在所有地方都用上了，因为好处总是有代价的。

节点在做出决定之前对提议进行投票的过程是一种同步复制。如“同步与异步复制”中所述，通常数据库会配置为异步复制模式。在这种配置中发生故障切换时，一些已经提交的数据可能会丢失 —— 但是为了获得更好的性能，许多人选择接受这种风险。

共识系统总是需要严格多数来运转。这意味着你至少需要三个节点才能容忍单节点故障（其余两个构成多数），或者至少有五个节点来容忍两个节点发生故障（其余三个构成多数）。如果网络故障切断了某些节点同其他节点的连接，则只有多数节点所在的网络可以继续工作，其余部分将被阻塞（参阅“线性一致性的代价”）。

大多数共识算法假定参与投票的节点是固定的集合，这意味着你不能简单的在集群中添加或删除节点。共识算法的动态成员扩展（**dynamic membership extension**）允许集群中的节点集随时间推移而变化，但是它们比静态成员算法要难理解得多。

共识系统通常依靠超时来检测失效的节点。在网络延迟高度变化的环境中，特别是在地理上散布的系统中，经常发生一个节点由于暂时的网络问题，错误地认为领导者已经失效。虽然这种错误不会损害安全属性，但频繁的领导者选举会导致糟糕的性能表现，因系统最后可能花在权力倾扎上的时间要比花在建设性工作的多得多。

有时共识算法对网络问题特别敏感。例如Raft已被证明存在让人不悦的极端情况【106】：如果整个网络工作正常，但只有一条特定的网络连接一直不可靠，Raft可能会进入领导频繁二人转的局面，或者当前领导者不断被迫辞职以致系统实质上毫无进展。其他一致性算法也存在类似的问题，而设计能健壮应对不可靠网络的算法仍然是一个开放的研究问题。

成员与协调服务

像ZooKeeper或etcd这样的项目通常被描述为“分布式键值存储”或“协调与配置服务”。这种服务的API看起来非常像数据库：你可以读写给定键的值，并遍历键。所以如果它们基本上算是数据库的话，为什么它们要把工夫全花在实现一个共识算法上呢？是什么使它们区别于其他任意类型的数据库？

为了理解这一点，简单了解如何使用ZooKeeper这类服务是很有帮助的。作为应用开发人员，你很少需要直接使用ZooKeeper，因为它实际上不适合当成通用数据库来用。更有可能的是，你会通过其他项目间接依赖它，例如HBase，Hadoop YARN，OpenStack Nova和Kafka都依赖ZooKeeper在后台运行。这些项目从它那里得到了什么？

ZooKeeper和etcd被设计为容纳少量完全可以放在内存中的数据（虽然它们仍然会写入磁盘以保证持久性），所以你不会想着把所有应用数据放到这里。这些少量数据会通过容错的全序广播算法复制到所有节点上。正如前面所讨论的那样，数据库复制需要的就是全序广播：如果每条消息代表对数据库的写入，则以相同的顺序应用相同的写入操作可以使副本之间保持一致。

ZooKeeper模仿了Google的Chubby锁服务【14,98】，不仅实现了全序广播（因此也实现了共识），而且还构建了一组有趣的其他特性，这些特性在构建分布式系统时变得特别有用：

线性一致性的原子操作

使用原子CAS操作可以实现锁：如果多个节点同时尝试执行相同的操作，只有一个节点会成功。共识协议保证了操作的原子性和线性一致性，即使节点发生故障或网络在任意时刻中断。分布式锁通常以租约（**lease**）的形式实现，租约有一个到期时间，以便在客户端失效的情况下最终能被释放（参阅“进程暂停”）。

操作的全序排序

如“领导者与锁定”中所述，当某个资源受到锁或租约的保护时，你需要一个防护令牌来防止客户端在进程暂停的情况下彼此冲突。防护令牌是每次锁被获取时单调增加的数字。ZooKeeper通过全局排序操作来提供这个功能，它为每个操作提供一个单调递增的事务ID（`zxid`）和版本号（`cversion`）【15】。

失效检测

客户端在ZooKeeper服务器上维护一个长期会话，客户端和服务器周期性地交换心跳包来检查节点是否还活着。即使连接暂时中断，或者ZooKeeper节点失效，会话仍保持在活跃状态。但如果心跳停止的持续时间超出会话超时，ZooKeeper会宣告该会话已死亡。当会话超时（ZooKeeper调用这些临时节点）时，会话持有的任何锁都可以配置为自动释放（ZooKeeper称之为临时节点（**ephemeral nodes**））。

变更通知

客户端不仅可以读取其他客户端创建的锁和值，还可以监听它们的变更。因此，客户端可以知道另一个客户端何时加入集群（基于新客户端写入ZooKeeper的值），或发生故障（因其会话超时，而其临时节点消失）。通过订阅通知，客户端不用再通过频繁轮询的方式来找出变更。

在这些功能中，只有线性一致的原子操作才真的需要共识。但正是这些功能的组合，使得像ZooKeeper这样的系统在分布式协调中非常有用。

将工作分配给节点

ZooKeeper/Chubby模型运行良好的一个例子是，如果你有几个进程实例或服务，需要选择其中一个实例作为主库或首选服务。如果领导者失败，其他节点之一应该接管。这对单主数据库当然非常实用，但对作业调度程序和类似的有状态系统也很好用。

另一个例子是，当你有一些分区资源（数据库，消息流，文件存储，分布式Actor系统等），并需要决定将哪个分区分配给哪个节点时。当新节点加入集群时，需要将某些分区从现有节点移动到新节点，以便重新平衡负载（参阅“重新平衡分区”）。当节点被移除或失效时，其他节点需要接管失效节点的工作。

这类任务可以通过在ZooKeeper中明智地使用原子操作，临时节点与通知来实现。如果设计得当，这种方法允许应用自动从故障中恢复而无需人工干预。不过这并不容易，尽管已经有不少在ZooKeeper客户端API基础之上提供更高层工具的库，例如Apache Curator 【17】。但它仍然要比尝试从头实现必要的共识算法要好得多，这样的尝试鲜有成功记录【107】。

应用最初只能在单个节点上运行，但最终可能会增长到数千个节点。试图在如此之多的节点上进行多数投票将是非常低效的。相反，ZooKeeper在固定数量的节点（通常是三到五个）上运行，并在这些节点之间执行其多数票，同时支持潜在的大量客户端。因此，ZooKeeper提供了一种将协调节点（共识，操作排序和故障检测）的一些工作“外包”到外部服务的方式。

通常，由ZooKeeper管理的数据的类型变化十分缓慢：代表“分区 7 中的节点运行在 `10.1.1.23` 上”的信息可能会在几分钟或几小时的时间内发生变化。它不是用来存储应用程序的运行时状态的，每秒可能会改变数千甚至数百万次。如果应用程序状态需要从一个节点复制到另一个节点，则可以使用其他工具（如Apache BookKeeper 【108】）。

服务发现

ZooKeeper，etcd和Consul也经常用于服务发现——也就是找出你需要连接到哪个IP地址才能到达特定的服务。在云数据中心环境中，虚拟机连续来去常见，你通常不会事先知道服务的IP地址。相反，你可以配置你的服务，使其在启动时注册服务注册表中的网络端点，然后可以由其他服务找到它们。

但是，服务发现是否需要达成共识还不太清楚。 DNS是查找服务名称的IP地址的传统方式，它使用多层缓存来实现良好的性能和可用性。从DNS读取是绝对不线性一致性的，如果DNS查询的结果有点陈旧，通常不会有问题【109】。 DNS对网络中断的可靠性和可靠性更为重

要。

尽管服务发现并不需要共识，但领导者选举却是如此。因此，如果你的共识系统已经知道领导是谁，那么也可以使用这些信息来帮助其他服务发现领导是谁。为此，一些共识系统支持只读缓存副本。这些副本异步接收共识算法所有决策的日志，但不主动参与投票。因此，它们能够提供不需要线性一致性的读取请求。

成员服务

ZooKeeper和它的小伙伴们可以看作是成员服务研究的悠久历史的一部分，这个历史可以追溯到20世纪80年代，并且对建立高度可靠的系统（例如空中交通管制）非常重要【110】。

成员资格服务确定哪些节点当前处于活动状态并且是群集的活动成员。正如我们在第8章中看到的那样，由于无限的网络延迟，无法可靠地检测到另一个节点是否发生故障。但是，如果你通过一致的方式进行故障检测，那么节点可以就哪些节点应该被认为是存在或不存在达成一致。

即使它确实存在，仍然可能发生一个节点被共识错误地宣告死亡。但是对于一个系统来说，在哪些节点构成当前的成员关系方面是非常有用的。例如，选择领导者可能意味着简单地选择当前成员中编号最小的成员，但如果不同的节点对现有成员的成员有不同意见，则这种方法将不起作用。

本章小结

在本章中，我们从几个不同的角度审视了关于一致性与共识的话题。我们深入研究了线性一致性（一种流行的一致性模型）：其目标是使多副本数据看起来好像只有一个副本一样，并使其上所有操作都原子性地生效。虽然线性一致性因为简单易懂而很吸引人 —— 它使数据库表现的好像单线程程序中的一个变量一样，但它有着速度缓慢的缺点，特别是在网络延迟很大的环境中。

我们还探讨了因果性，因果性对系统中的事件施加了顺序（什么发生在什么之前，基于因与果）。与线性一致不同，线性一致性将所有操作放在单一的全序时间线中，因果一致性为我们提供了一个较弱的一致性模型：某些事件可以是并发的，所以版本历史就像是一条不断分叉与合并的时间线。因果一致性没有线性一致性的协调开销，而且对网络问题的敏感性要低得多。

但即使捕获到因果顺序（例如使用兰伯特时间戳），我们发现有些事情也不能通过这种方式实现：在“光有时间戳排序还不够”一节的例子中，我们需要确保用户名是唯一的，并拒绝同一用户名的其他并发注册。如果一个节点要通过注册，则需要知道其他的节点没有在并发抢注同一用户名的过程中。这个问题引领我们走向共识。

我们看到，达成共识意味着以这样一种方式决定某件事：所有节点一致同意所做决定，且这一决定不可撤销。通过深入挖掘，结果我们发现很广泛的一系列问题实际上都可以归结为共识问题，并且彼此等价（从这个意义上来讲，如果你有其中之一的解决方案，就可以轻易将它转换为其他问题的解决方案）。这些等价的问题包括：

线性一致性的***CAS***寄存器

寄存器需要基于当前值是否等于操作给出的参数，原子地决定是否设置新值。

原子事务提交

数据库必须决定是否提交或中止分布式事务。

全序广播

消息系统必须决定传递消息的顺序。

锁和租约

当几个客户端争抢锁或租约时，由锁来决定哪个客户端成功获得锁。

成员/协调服务

给定某种故障检测器（例如超时），系统必须决定哪些节点活着，哪些节点因为会话超时需要被宣告死亡。

唯一性约束

当多个事务同时尝试使用相同的键创建冲突记录时，约束必须决定哪一个被允许，哪些因为违反约束而失败。

如果你只有一个节点，或者你愿意将决策的权能分配给单个节点，所有这些事都很简单。这就是在单领导者数据库中发生的事情：所有决策权归属于领导者，这就是为什么这样的数据库能够提供线性一致的操作，唯一性约束，完全有序的复制日志，以及更多。

但如果该领导者失效，或者如果网络中断导致领导者不可达，这样的系统就无法取得任何进展。应对这种情况可以有三种方法：

1. 等待领导者恢复，接受系统将在这段时间阻塞的事实。许多XA/JTA事务协调者选择这个选项。这种方法并不能完全达成共识，因为它不能满足终止属性的要求：如果领导者续命失败，系统可能会永久阻塞。
2. 人工故障切换，让人类选择一个新的领导者节点，并重新配置系统使之生效，许多关系型数据库都采用这种方方式。这是一种来自“天意”的共识 —— 由计算机系统之外的运维人员做出决定。故障切换的速度受到人类行动速度的限制，通常要比计算机慢（得多）。
3. 使用算法自动选择一个新的领导者。这种方法需要一种共识算法，使用成熟的算法来正确处理恶劣的网络条件是明智之举【107】。

尽管单领导者数据库可以提供线性一致性，且无需对每个写操作都执行共识算法，但共识对于保持及变更领导权仍然是必须的。因此从某种意义上说，使用单个领导者不过是“缓兵之计”：共识仍然是需要的，只是在另一个地方，而且没那么频繁。好消息是，容错的共识算法与容错的共识系统是存在的，我们在本章中简要地讨论了它们。

像ZooKeeper这样的工具为应用提供了“外包”的共识、故障检测和成员服务。它们扮演了重要的角色，虽说使用不易，但总比自己去开发一个能经受第8章中所有问题考验的算法要好得多。如果你发现自己想要解决的问题可以归结为共识，并且希望它能容错，使用一个类似ZooKeeper的东西是明智之举。

尽管如此，并不是所有系统都需要共识：例如，无领导者复制和多领导者复制系统通常不会使用全局的共识。这些系统中出现的冲突（参见“处理冲突”）正是不同领导者之间没有达成共识的结果，但也这并没有关系：也许我们只是需要接受没有线性一致性的事实，并学会更好地与具有分支与合并版本历史的数据打交道。

本章引用了大量关于分布式系统理论的研究。虽然理论论文和证明并不总是容易理解，有时也会做出不切实际的假设，但它们对于指导这一领域的实践有着极其重要的价值：它们帮助我们推理什么可以做，什么不可以做，帮助我们找到反直觉的分布式系统缺陷。如果你有时间，这些参考资料值得探索。

这里已经到了本书第二部分的末尾，第二部介绍了复制（第5章），分区（第6章），事务（第7章），分布式系统的故障模型（第8章）以及最后的一致性与共识（第9章）。现在我们已经奠定了扎实的理论基础，我们将在第三部分再次转向更实际的系统，并讨论如何使用异构的组件积木块构建强大的应用。

参考文献

1. Peter Bailis and Ali Ghodsi: “Eventual Consistency Today: Limitations, Extensions, and Beyond,” *ACM Queue*, volume 11, number 3, pages 55-63, March 2013. doi:10.1145/2460276.2462076
2. Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin: “Consistency, Availability, and Convergence,” University of Texas at Austin, Department of Computer Science, Tech Report UTCS TR-11-22, May 2011.
3. Alex Scotti: “Adventures in Building Your Own Database,” at *All Your Base*, November 2015.
4. Peter Bailis, Aaron Davidson, Alan Fekete, et al.: “Highly Available Transactions: Virtues and Limitations,” at *40th International Conference on Very Large Data Bases* (VLDB), September 2014. Extended version published as pre-print arXiv:1302.0309 [cs.DB].
5. Paolo Viotti and Marko Vukolić: “Consistency in Non-Transactional Distributed Storage Systems,” arXiv:1512.00168, 12 April 2016.

6. Maurice P. Herlihy and Jeannette M. Wing: "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 12, number 3, pages 463–492, July 1990. doi:10.1145/78969.78972

7. Leslie Lamport: "On interprocess communication," *Distributed Computing*, volume 1, number 2, pages 77–101, June 1986. doi:10.1007/BF01786228

8. David K. Gifford: "Information Storage in a Decentralized Computer System," Xerox Palo Alto Research Centers, CSL-81-8, June 1981.

9. Martin Kleppmann: "Please Stop Calling Databases CP or AP," *martin.kleppmann.com*, May 11, 2015.

10. Kyle Kingsbury: "Call Me Maybe: MongoDB Stale Reads," *aphyr.com*, April 20, 2015.

11. Kyle Kingsbury: "Computational Techniques in Knossos," *aphyr.com*, May 17, 2014.

12. Peter Bailis: "Linearizability Versus Serializability," *bailis.org*, September 24, 2014.

13. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at *research.microsoft.com*.

14. Mike Burrows: "The Chubby Lock Service for Loosely-Coupled Distributed Systems," at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

15. Flavio P. Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN: 978-1-449-36130-3

16. "etcd 2.0.12 Documentation," CoreOS, Inc., 2015.

17. "Apache Curator," Apache Software Foundation, *curator.apache.org*, 2015.

18. Morali Vallath: *Oracle 10g RAC Grid, Services & Clustering*. Elsevier Digital Press, 2006. ISBN: 978-1-555-58321-7

19. Peter Bailis, Alan Fekete, Michael J Franklin, et al.: "Coordination-Avoiding Database Systems," *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.

20. Kyle Kingsbury: "Call Me Maybe: etcd and Consul," *aphyr.com*, June 9, 2014.

21. Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini: "Zab: High-Performance Broadcast for Primary-Backup Systems," at *41st IEEE International Conference on Dependable Systems and Networks* (DSN), June 2011. doi:10.1109/DSN.2011.5958223

22. Diego Ongaro and John K. Ousterhout: “In Search of an Understandable Consensus Algorithm (Extended Version),” at *USENIX Annual Technical Conference* (ATC), June 2014.

23. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev: “Sharing Memory Robustly in Message-Passing Systems,” *Journal of the ACM*, volume 42, number 1, pages 124–142, January 1995. doi:10.1145/200836.200869

24. Nancy Lynch and Alex Shvartsman: “Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts,” at *27th Annual International Symposium on Fault-Tolerant Computing* (FTCS), June 1997. doi:10.1109/FTCS.1997.614100

25. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, 2nd edition. Springer, 2011. ISBN: 978-3-642-15259-7, doi:10.1007/978-3-642-15260-3

26. Sam Elliott, Mark Allen, and Martin Kleppmann: personal communication, thread on *twitter.com*, October 15, 2015.

27. Niklas Ekström, Mikhail Panchenko, and Jonathan Ellis: “Possible Issue with Read Repair?,” email thread on *cassandra-dev* mailing list, October 2012.

28. Maurice P. Herlihy: “Wait-Free Synchronization,” *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 13, number 1, pages 124–149, January 1991. doi:10.1145/114005.102808

29. Armando Fox and Eric A. Brewer: “Harvest, Yield, and Scalable Tolerant Systems,” at *7th Workshop on Hot Topics in Operating Systems* (HotOS), March 1999. doi:10.1109/HOTOS.1999.798396

30. Seth Gilbert and Nancy Lynch: “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” *ACM SIGACT News*, volume 33, number 2, pages 51–59, June 2002. doi:10.1145/564585.564601

31. Seth Gilbert and Nancy Lynch: “Perspectives on the CAP Theorem,” *IEEE Computer Magazine*, volume 45, number 2, pages 30–36, February 2012. doi:10.1109/MC.2011.389

32. Eric A. Brewer: “CAP Twelve Years Later: How the 'Rules' Have Changed,” *IEEE Computer Magazine*, volume 45, number 2, pages 23–29, February 2012. doi:10.1109/MC.2012.37

33. Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen: “Consistency in Partitioned Networks,” *ACM Computing Surveys*, volume 17, number 3, pages 341–370, September 1985. doi:10.1145/5505.5508

34. Paul R. Johnson and Robert H. Thomas: "RFC 677: The Maintenance of Duplicate Databases," Network Working Group, January 27, 1975.

35. Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: "Notes on Distributed Databases," IBM Research, Research Report RJ2571(33471), July 1979.

36. Michael J. Fischer and Alan Michael: "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," at *1st ACM Symposium on Principles of Database Systems* (PODS), March 1982. doi:10.1145/588111.588124

37. Eric A. Brewer: "NoSQL: Past, Present, Future," at *QCon San Francisco*, November 2012.

38. Henry Robinson: "CAP Confusion: Problems with 'Partition Tolerance,'" *blog.cloudera.com*, April 26, 2010.

39. Adrian Cockcroft: "Migrating to Microservices," at *QCon London*, March 2014.

40. Martin Kleppmann: "A Critique of the CAP Theorem," arXiv:1509.05393, September 17, 2015.

41. Nancy A. Lynch: "A Hundred Impossibility Proofs for Distributed Computing," at *8th ACM Symposium on Principles of Distributed Computing* (PODC), August 1989. doi:10.1145/72981.72982

42. Hagit Attiya, Faith Ellen, and Adam Morrison: "Limitations of Highly-Available Eventually-Consistent Data Stores](http://www.cs.technion.ac.il/people/mad/online-publications/podc2015-replds.pdf)," at *ACM Symposium on Principles of Distributed Computing* (PODC), July 2015. doi:10.1145/2767386.2767419

43. Peter Sewell, Susmit Sarkar, Scott Owens, et al.: "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM*, volume 53, number 7, pages 89–97, July 2010. doi:10.1145/1785414.1785443

44. Martin Thompson: "Memory Barriers/Fences," *mechanical-sympathy.blogspot.co.uk*, July 24, 2011.

45. Ulrich Drepper: "What Every Programmer Should Know About Memory," *akkadia.org*, November 21, 2007.

46. Daniel J. Abadi: "Consistency Tradeoffs in Modern Distributed Database System Design," *IEEE Computer Magazine*, volume 45, number 2, pages 37–42, February 2012. doi:10.1109/MC.2012.33

47. Hagit Attiya and Jennifer L. Welch: "Sequential Consistency Versus Linearizability," *ACM Transactions on Computer Systems* (TOCS), volume 12, number 2, pages 91–122, May 1994. doi:10.1145/176575.176576

48. Mustaque Ahamad, Gil Neiger, James E. Burns, et al.: "Causal Memory: Definitions, Implementation, and Programming," *Distributed Computing*, volume 9, number 1, pages 37–49, March 1995. doi:10.1007/BF01784241

49. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen: "Stronger Semantics for Low-Latency Geo-Replicated Storage," at *10th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2013.

50. Marek Zawirski, Annette Bieniusa, Valter Balegas, et al.: "SwiftCloud: Fault-Tolerant Geo-Replication Integrated All the Way to the Client Machine," INRIA Research Report 8347, August 2013.

51. Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica: "Bolt-on Causal Consistency," at *ACM International Conference on Management of Data* (SIGMOD), June 2013.

52. Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: "Challenges to Adopting Stronger Consistency at Scale," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

53. Peter Bailis: "Causality Is Expensive (and What to Do About It)," *bailis.org*, February 5, 2014.

54. Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte: "Concise Server-Wide Causality Management for Eventually Consistent Data Stores," at *15th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS), June 2015. doi:10.1007/978-3-319-19129-4_6

55. Rob Conery: "A Better ID Generator for PostgreSQL," *rob.conery.io*, May 29, 2014.

56. Leslie Lamport: "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563

57. Xavier Défago, André Schiper, and Péter Urbán: "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Surveys*, volume 36, number 4, pages 372–421, December 2004. doi:10.1145/1041680.1041682

58. Hagit Attiya and Jennifer Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd edition. John Wiley & Sons, 2004. ISBN: 978-0-471-45324-6, doi:10.1002/0471478210

59. Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, et al.: "CORFU: A Shared Log Design for Flash Clusters," at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

60. Fred B. Schneider: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, volume 22, number 4, pages 299–319, December 1990.

61. Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, et al.: "Calvin: Fast Distributed Transactions for Partitioned Database Systems," at *ACM International Conference on Management of Data* (SIGMOD), May 2012.

62. Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, et al.: "Tango: Distributed Data Structures over a Shared Log," at *24th ACM Symposium on Operating Systems Principles* (SOSP), November 2013. doi:10.1145/2517349.2522732

63. Robbert van Renesse and Fred B. Schneider: "Chain Replication for Supporting High Throughput and Availability," at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

64. Leslie Lamport: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, volume 28, number 9, pages 690–691, September 1979. doi:10.1109/TC.1979.1675439

65. Enis Söztutar, Devaraj Das, and Carter Shanklin: "Apache HBase High Availability at the Next Level," *hortonworks.com*, January 22, 2015.

66. Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, et al.: "PNUTS: Yahoo!'s Hosted Data Serving Platform," at *34th International Conference on Very Large Data Bases* (VLDB), August 2008. doi:10.14778/1454159.1454167

67. Tushar Deepak Chandra and Sam Toueg: "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, volume 43, number 2, pages 225–267, March 1996. doi:10.1145/226643.226647

68. Michael J. Fischer, Nancy Lynch, and Michael S. Paterson: "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, volume 32, number 2, pages 374–382, April 1985. doi:10.1145/3149.214121

69. Michael Ben-Or: "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," at *2nd ACM Symposium on Principles of Distributed Computing* (PODC), August 1983. doi:10.1145/800221.806707

70. Jim N. Gray and Leslie Lamport: “Consensus on Transaction Commit,” *ACM Transactions on Database Systems* (TODS), volume 31, number 1, pages 133–160, March 2006. doi:10.1145/1132863.1132867

71. Rachid Guerraoui: “Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus,” at *9th International Workshop on Distributed Algorithms* (WDAG), September 1995. doi:10.1007/BFb0022140

72. Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications,” at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.

73. Jim Gray: “The Transaction Concept: Virtues and Limitations,” at *7th International Conference on Very Large Data Bases* (VLDB), September 1981.

74. Hector Garcia-Molina and Kenneth Salem: “Sagas,” at *ACM International Conference on Management of Data* (SIGMOD), May 1987. doi:10.1145/38713.38742

75. C. Mohan, Bruce G. Lindsay, and Ron Obermarck: “Transaction Management in the R* Distributed Database Management System,” *ACM Transactions on Database Systems*, volume 11, number 4, pages 378–396, December 1986. doi:10.1145/7239.7266

76. “Distributed Transaction Processing: The XA Specification,” X/Open Company Ltd., Technical Standard XO/CAE/91/300, December 1991. ISBN: 978-1-872-63024-3

77. Mike Spille: “XA Exposed, Part II,” *jroller.com*, April 3, 2004.

78. Ivan Silva Neto and Francisco Reverbel: “Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction,” at *7th IEEE/ACIS International Conference on Computer and Information Science* (ICIS), May 2008. doi:10.1109/ICIS.2008.75

79. James E. Johnson, David E. Langworthy, Leslie Lamport, and Friedrich H. Vogt: “Formal Specification of a Web Services Protocol,” at *1st International Workshop on Web Services and Formal Methods* (WS-FM), February 2004. doi:10.1016/j.entcs.2004.02.022

80. Dale Skeen: “Nonblocking Commit Protocols,” at *ACM International Conference on Management of Data* (SIGMOD), April 1981. doi:10.1145/582318.582339

81. Gregor Hohpe: “Your Coffee Shop Doesn’t Use Two-Phase Commit,” *IEEE Software*, volume 22, number 2, pages 64–66, March 2005. doi:10.1109/MS.2005.52

82. Pat Helland: “Life Beyond Distributed Transactions: An Apostate’s Opinion,” at *3rd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2007.

83. Jonathan Oliver: "My Beef with MSDTC and Two-Phase Commits," *blog.jonathanoliver.com*, April 4, 2011.

84. Oren Eini (Ahende Rahien): "The Fallacy of Distributed Transactions," *ayende.com*, July 17, 2014.

85. Clemens Vasters: "Transactions in Windows Azure (with Service Bus) – An Email Discussion," *vasters.com*, July 30, 2012.

86. "Understanding Transactionality in Azure," NServiceBus Documentation, Particular Software, 2015.

87. Randy Wigginton, Ryan Lowe, Marcos Albe, and Fernando Ipar: "Distributed Transactions in MySQL," at *MySQL Conference and Expo*, April 2013.

88. Mike Spille: "XA Exposed, Part I," *jroller.com*, April 3, 2004.

89. Ajmer Dhariwal: "Orphaned MSDTC Transactions (-2 spids)," *eraofdata.com*, December 12, 2008.

90. Paul Randal: "Real World Story of DBCC PAGE Saving the Day," *sqlskills.com*, June 19, 2013.

91. "in-doubt xact resolution Server Configuration Option," SQL Server 2016 documentation, Microsoft, Inc., 2016.

92. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:10.1145/42282.42283

93. Miguel Castro and Barbara H. Liskov: "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems*, volume 20, number 4, pages 396–461, November 2002. doi:10.1145/571637.571640

94. Brian M. Oki and Barbara H. Liskov: "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," at *7th ACM Symposium on Principles of Distributed Computing* (PODC), August 1988. doi:10.1145/62546.62549

95. Barbara H. Liskov and James Cowling: "Viewstamped Replication Revisited," Massachusetts Institute of Technology, Tech Report MIT-CSAIL-TR-2012-021, July 2012.

96. Leslie Lamport: "The Part-Time Parliament," *ACM Transactions on Computer Systems*, volume 16, number 2, pages 133–169, May 1998. doi:10.1145/279227.279229

97. Leslie Lamport: "Paxos Made Simple," *ACM SIGACT News*, volume 32, number 4, pages 51–58, December 2001.

98. Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone: "Paxos Made Live – An Engineering Perspective," at *26th ACM Symposium on Principles of Distributed Computing* (PODC), June 2007.

99. Robbert van Renesse: "Paxos Made Moderately Complex," *cs.cornell.edu*, March 2011.

00. Diego Ongaro: "Consensus: Bridging Theory and Practice," PhD Thesis, Stanford University, August 2014.

01. Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft: "Raft Refloated: Do We Have Consensus?," *ACM SIGOPS Operating Systems Review*, volume 49, number 1, pages 12–21, January 2015. doi:10.1145/2723872.2723876

02. André Medeiros: "ZooKeeper's Atomic Broadcast Protocol: Theory and Practice," Aalto University School of Science, March 20, 2012.

03. Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider: "Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab," *IEEE Transactions on Dependable and Secure Computing*, volume 12, number 4, pages 472–484, September 2014. doi:10.1109/TDSC.2014.2355848

04. Will Portnoy: "Lessons Learned from Implementing Paxos," *blog.willportnoy.com*, June 14, 2012.

05. Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: "Flexible Paxos: Quorum Intersection Revisited," *arXiv:1608.06696*, August 24, 2016.

06. Heidi Howard and Jon Crowcroft: "Coracle: Evaluating Consensus at the Internet Edge," at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), August 2015. doi:10.1145/2829988.2790010

07. Kyle Kingsbury: "Call Me Maybe: Elasticsearch 1.5.0," *aphyr.com*, April 27, 2015.

08. Ivan Kelly: "BookKeeper Tutorial," *github.com*, October 2014.

09. Camille Fournier: "Consensus Systems for the Skeptical Architect," at *Craft Conference*, Budapest, Hungary, April 2015.

10. Kenneth P. Birman: "A History of the Virtual Synchrony Replication Model," in *Replication: Theory and Practice*, Springer LNCS volume 5959, chapter 6, pages 91–120, 2010. ISBN: 978-3-642-11293-5, doi:10.1007/978-3-642-11294-2_6

第三部分：衍生数据

在本书的第一部分和第二部分中，我们自底向上地把所有关于分布式数据库的主要考量都过了一遍。从数据在磁盘上的布局，一直到出现故障时分布式系统一致性的局限。但所有的讨论都假定了应用中只用了一种数据库。

现实世界中的数据系统往往更为复杂。大型应用程序经常需要以多种方式访问和处理数据，没有一个数据库可以同时满足所有这些不同的需求。因此应用程序通常组合使用多种组件：数据存储，索引，缓存，分析系统，等等，并实现在这些组件中移动数据的机制。

本书的最后一部分，会研究将多个不同数据系统（可能有着不同数据模型，并针对不同的访问模式进行优化）集成为一个协调一致的应用架构时，会遇到的问题。软件供应商经常会忽略这一方面的生态建设，并声称他们的产品能够满足你的所有需求。在现实世界中，集成不同的系统是实际应用中最重要的事情之一。

记录和衍生数据系统

从高层次上看，存储和处理数据的系统可以分为两大类：

记录系统（**System of record**）

记录系统，也被称为真相源（**source of truth**），持有数据的权威版本。当新的数据进入时（例如，用户输入）首先会记录在这里。每个事实正正好好表示一次（表示通常是标准化的（**normalized**））。如果其他系统和记录系统之间存在任何差异，那么记录系统中的值是正确的（根据定义）。

衍生数据系统（**Derived data systems**）

衍生系统中的数据，通常是另一个系统中的现有数据以某种方式进行转换或处理的结果。如果丢失衍生数据，可以从原始来源重新创建。典型的例子是缓存（**cache**）：如果数据在缓存中，就可以由缓存提供服务；如果缓存不包含所需数据，则降级由底层数据库提供。非规范化的值，索引和物化视图亦属此类。在推荐系统中，预测汇总数据通常衍生自用户日志。

从技术上讲，衍生数据是冗余的（**redundant**），因为它重复了已有的信息。但是衍生数据对于获得良好的只读查询性能通常是至关重要的。它通常是非规范化的。可以从单个源头衍生出多个不同的数据集，使你能从不同的“视角”洞察数据。

并不是所有的系统都在其架构中明确区分记录系统和衍生数据系统，但是这是一种有用的区分方式，因为它明确了系统中的数据流：系统的哪一部分具有哪些输入和哪些输出，以及它们如何相互依赖。

大多数数据库，存储引擎和查询语言，本质上既不是记录系统也不是衍生系统。数据库只是一个工具：如何使用它取决于你自己。记录系统和衍生数据系统之间的区别不在于工具，而在于应用程序中的使用方式。

通过梳理数据的派衍生关系，可以清楚地理解一个令人困惑的系统架构。这将贯穿本书的这一部分。

章节概述

我们将从第十章开始，研究例如MapReduce这样面向批处理（**batch-oriented**）的数据流系统。对于建设大规模数据系统，我们将看到，它们提供了优秀的工具和思想。第十一章将把这些思想应用到流式数据（**data streams**）中，使我们能用更低的延迟完成同样的任务。第十二章将对本书进行总结，探讨如何使用这些工具来构建可靠，可扩展和可维护的应用。

索引

上一章	目录	下一章
第九章：一致性与共识	设计数据密集型应用	第十章：批处理

10. 批处理



> 带有太强个人色彩的系统无法成功。当第一版健壮的设计完成时，不同的人们以自己的方式来测试时，真正的考验才开始。
>
> ——高德纳

[TOC]

在本书的前两部分中，我们讨论了很多关于请求（**requests**）和查询（**queries**）以及相应的响应（**response**）或结果（**results**）。在许多现代数据系统中都假设采用这种数据处理方式：你要求某些东西，或者发送指令，一段时间后（希望）系统会给你一个答案。数据库，缓存，搜索索引，Web服务器以及其他许多系统都以这种方式工作。

在这类在线（**online**）系统中，无论是浏览器请求页面还是调用远程API的服务，我们通常都假设请求是由人类用户触发的，且用户正在等待响应。他们不必等太久，所以我们非常重视这些系统的响应时间（参阅“描述性能”）。

Web和越来越多的基于HTTP/REST的API使交互的请求/响应风格变得如此普遍，以至于很容易将其视为理所当然。但我们应该记住，这不是构建系统的唯一方式，其他方法也有其优点。我们来区分三种不同类型的系统：

服务（在线系统）

服务等待客户的请求或指令到达。当收到一个，服务试图尽快处理它，并发回一个响应。响应时间通常是服务性能的主要衡量指标，可用性通常非常重要（如果客户端无法访问服务，用户可能会收到错误消息）。

批处理系统（离线系统）

一个批处理系统需要大量的输入数据，运行一个工作来处理它，并产生一些输出数据。工作往往需要一段时间（从几分钟到几天），所以通常不会有用户等待工作完成。相反，批量作业通常会定期运行（例如，每天一次）。批处理作业的主要性能衡量标准通常是吞吐量（通过特定大小的输入数据集所需的时间）。我们将在本章中讨论批处理。

流处理系统（近实时系统）

流处理是在线和离线/批处理之间的一个地方（所以有时候被称为近实时或近线处理）。像批处理系统一样，流处理器消耗输入并产生输出（而不是响应请求）。但是，流式作业在事件发生后不久就会对事件进行操作，而批处理作业则使用固定的一组输入数据进行操作。这种差异使流处理系统比等效的批处理系统具有更低的延迟。由于流处理基于批处理，我们将在第11章讨论它。

正如我们将在本章中看到的那样，批量处理是构建可靠，可扩展和可维护应用程序的重要组成部分。例如，2004年发布的批处理算法Map-Reduce（可能过热地）被称为“使得Google具有如此大规模可扩展性的算法”【2】。随后在各种开源数据系统中实现，包括Hadoop，CouchDB和MongoDB。

与多年前为数据仓库开发的并行处理系统【3,4】相比，MapReduce是一个相当低级别的编程模型，但它对于在商用硬件上的处理规模上迈出了重要的一步。。虽然MapReduce的重要性正在下降【5】，但它仍然值得理解，因为它提供了一副批处理为什么，以及如何有用的清晰画面。

实际上，批处理是一种非常古老的计算形式。早在可编程数字计算机诞生之前，打孔卡制表机（例如1890年美国人口普查【6】中使用的霍尔里斯机）实现了半机械化的批处理形式，以计算来自大量输入的汇总统计量。 Map-Reduce与1940年代和1950年代广泛用于商业数据处理的机电IBM卡片分类机器有着惊人的相似之处【7】。像往常一样，历史总是在不断重复自己。

在本章中，我们将看看MapReduce和其他一些批处理算法和框架，并探讨它们在现代数据系统中的使用方式。但首先，我们将从如何使用标准Unix工具进行数据处理开始。即使你已经熟悉了它们，也值得重温一下Unix哲学，因为从Unix的想法和经验教训能转移到大规模，异构的分布式数据系统。

使用Unix工具的批处理

我们从一个简单的例子开始。假设你有一台Web服务器，每次提供请求时都会在日志文件中附加一行。例如，使用nginx默认访问日志格式，日志的一行可能如下所示：

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000] "GET /css/typography.css HTTP/1.1"
200 3377 "http://martin.kleppmann.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5
)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115 Safari/537.36"
```

（实际上这只是一行，为了可读性拆分为多行）。这一行中有很多信息。为了解释它，你需要看一看日志格式的定义，如下所示：:

```
 $remote_addr - $remote_user [$time_local] "$request"
 $status $body_bytes_sent "$http_referer" "$http_user_agent"
```

因此，日志的这一行表明，在 `2015-02-27 17:55:11 UTC` ，服务器从客户端IP地址 `216.58.210.78` 接收到文件 `/css/typography.css` 的请求。用户没有被认证，所以 `$remote_user` 被设置为连字符（ `-` ）。响应状态是200（即请求成功），响应的大小是3377字节。网页浏览器是Chrome 40，并且它加载了该文件，因为它是在URL `http://martin.kleppmann.com/` 的页面中引用的。

分析简单日志

各种工具可以把这些日志文件，并产生漂亮的报告有关你的网站流量，但出于练习的目的，让我们使用基本的Unix工具自己写一个。 例如，假设你想在你的网站上找到五个最受欢迎的网页。 可以在Unix shell中这样做：[i]

> i. 有些人喜欢抬杠，认为 `cat` 这里并没有必要，因为输入文件可以直接作为awk的参数。但这种写法让流水线更显眼。 ↩

```
cat /var/log/nginx/access.log | #1
    awk '{print $7}' | #2
    sort             | #3
    uniq -c          | #4
    sort -r -n       | #5
    head -n 5          #6
```

1. 读取日志文件
2. 将每一行按空格分割成不同的字段，每行只输出第七个这样的字段，恰好是请求的URL。在我们的示例行中，这个请求URL是 `/css/typography.css` 。
3. 按字母顺序排列请求的URL列表。如果某个URL被请求过n次，那么排序后，该文件将包含连续n次重复的URL。
4. uniq命令通过检查两条相邻的行是否相同来过滤掉其输入中的重复行。 `-c` 选项告诉它也输出一个计数器：对于每个不同的URL，它会报告输入中出现该URL的次数。
5. 第二种排序按每行起始处的数字（ `-n` ）排序，这是请求URL的次数。然后以反向（ `-r` ）顺序返回结果，即首先以最大的数字返回结果。
6. 最后，头只输出输入的前五行（ `-n 5` ），并丢弃其余的。该系列命令的输出如下所示：

```
    4189 /favicon.ico
    3631 /2013/05/24/improving-security-of-ssh-private-keys.html
    2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
    1369 /
     915 /css/typography.css
```

尽管如果你不熟悉Unix工具，上面的命令行可能看起来有点模糊，但是它非常强大。它将在几秒钟内处理千兆字节的日志文件，你可以轻松修改分析以适应你的需求。例如，如果要从报告中省略CSS文件，请将 `awk` 参数更改为 `'$7 !~ /\.css$/ {print $7}'` 等等。

本书中没有空间来详细探索Unix工具，但是非常值得学习。令人惊讶的是，使用awk，sed，grep，sort，uniq和xargs的组合，可以在几分钟内完成许多数据分析，并且它们的表现令人惊讶地很好【8】。

命令链与自定义程序

除了Unix命令链，你还可以写一个简单的程序来做同样的事情。例如在Ruby中，它可能看起来像这样：

```
counts = Hash.new(0)         # 1
File.open('/var/log/nginx/access.log') do |file|
    file.each do |line|
        url = line.split【6】   # 2
        counts[url] += 1      # 3
    end
end

top5 = counts.map{|url, count| [count, url] }.sort.reverse[0...5] # 4
top5.each{|count, url| puts "#{count} #{url}" }                   # 5
```

1. `counts` 是一个哈希表，保存了我们浏览每个URL次数的计数器，计数器默认为零。

2. 对日志的每一行，从第七个空格分隔的字段提取URL（这里的数组索引是6，因为Ruby的数组索引从零开始）。
3. 增加日志当前行中URL的计数器。
4. 按计数器值（降序）对散列表内容进行排序，并取前五位。
5. 打印出前五个条目。

这个程序并不像Unix管道那样简洁，但是它的可读性很强，你喜欢哪一个属于口味的问题。但两者除了表面上的差异之外，执行流程也有很大差异，如果你在大文件上运行此分析，则会变得明显。

排序与内存中的聚合

Ruby脚本保存一个URL的内存哈希表，其中每个URL映射到它已经被看到的次数。 Unix流水线的例子没有这样一个哈希表，而是依赖于对一个URL列表进行排序，在这个URL列表中，同一个URL的多个发生只是简单的重复。

哪种方法更好？这取决于你有多少个不同的网址。对于大多数中小型网站，你可能可以适应所有不同的URL，并且可以为每个网址（例如1GB内存）提供一个计数器。在此示例中，作业的工作集（作业需要随机访问的内存量）仅取决于不同URL的数量：如果单个URL有一百万个日志条目，则散列中所需的空间表仍然只有一个URL加上计数器的大小。如果这个工作集足够小，那么内存散列表工作正常，甚至在笔记本电脑上也是如此。

另一方面，如果作业的工作集大于可用内存，则排序方法的优点是可以高效地使用磁盘。这与我们在第74页的“SSTables和LSM树”中讨论过的原理是一样的：数据块可以在内存中排序并作为段文件写入磁盘，然后多个排序的段可以合并为一个更大的排序文件。 归并排序具有在磁盘上运行良好的顺序访问模式。 （请记住，优化为顺序I/O是第3章中反复出现的主题。这里再次出现相同的模式。）

GNU Coreutils（Linux）中的排序实用程序通过溢出到磁盘自动处理大于内存的数据集，并自动并行排序跨多个CPU核心【9】。这意味着我们之前看到的简单的Unix命令链很容易扩展到大数据集，而不会耗尽内存。瓶颈可能是从磁盘读取输入文件的速度。

Unix哲学

我们可以非常容易地使用前一个例子中的一系列命令来分析日志文件，这并非巧合：事实上，这实际上是Unix的关键设计思想之一，且它今天仍然令人惊讶地关联。让我们更深入地研究一下，以便从Unix中借鉴一些想法【10】。

Unix管道的发明者道格·麦克罗伊（Doug McIlroy）在1964年首先描述了这种情况【11】：“当需要以另一种方式处理数据时，我们应该有一些连接程序的方法，比如[a] 。这也是I/O的方式。“管道类比困难了，连接程序和管道的想法成为了现在被称为**Unix**哲学的一部分 —— 一套在开发者中流行的设计原则。 Unix哲学在1978年描述如下【12,13】：

1. 让每个程序都做好一件事。要做一件新的工作，写一个新程序，而不是通过添加“功能”让

老程序复杂化。
2. 期待每个程序的输出成为另一个程序的输入。不要将无关信息混入输出。避免使用严格的列数据或二进制输入格式。不要坚持交互式输入。
3. 设计和构建软件，甚至是操作系统，要尽早尝试，最好在几周内完成。不要犹豫，扔掉笨拙的部分，重建它们。
4. 优先使用工具，而不是不熟练的帮助来减轻编程任务，即使必须曲线救国编写工具，并期望在用完后扔掉大部分。

这种方法 —— 自动化，快速原型设计，迭代式迭代，实验友好，将大型项目分解成可管理的块 —— 听起来非常像今天的敏捷开发和DevOps运动。奇怪的是，四十年来变化不大。

排序工具是一个很好的例子。它可以说是一个比大多数编程语言在其标准库（不会溢出到磁盘并且不使用多线程，即使是有益的）中更好的排序实现。然而，单独使用 `sort` 几乎没什么用。它只能与其他Unix工具（如 `uniq` ）结合使用。

像bash这样的Unix shell可以让我们轻松地将这些小程序组合成令人惊讶的强大数据处理作业。尽管这些程序中有很多是由不同人群编写的，但它们可以灵活地结合在一起。 Unix如何实现这种可组合性？

统一的接口

如果你希望一个程序的输出成为另一个程序的输入，那意味着这些程序必须使用相同的数据格式 —— 换句话说，一个兼容的接口。如果你希望能够将任何程序的输出连接到任何程序的输入，那意味着所有程序必须使用相同的I/O接口。

在Unix中，这种接口是一个文件（**file**）（更准确地说，是一个文件描述符（**file descriptor**））。一个文件只是一串有序的字节序列。因为这是一个非常简单的接口，所以可以使用相同的接口来表示许多不同的东西：文件系统上的真实文件，到另一个进程（Unix套接字，stdin，stdout）的通信通道，设备驱动程序（比如 `/dev/audio` 或 `/dev/lp0` ），表示TCP连接的套接字等等。很容易将这些视为理所当然的事情，但实际上这是非常出色的设计：这些非常不同的事物可以共享一个统一的接口，所以它们可以很容易地连接在一起[ii]。

> [ii]. 统一接口的另一个例子是URL和HTTP，这是Web的基础。 一个URL标识一个网站上的一个特定的东西（资源），你可以链接到任何其他网站的任何网址。 具有网络浏览器的用户因此可以通过跟随链接在网站之间无缝跳转，即使服务器可能由完全不相关的组织操作。 这个原则今天似乎很明显，但它是使网络取得今天成功的关键。 之前的系统并不是那么统一：例如，在公告板系统（BBS）时代，每个系统都有自己的电话号码和波特率配置。 从一个BBS到另一个BBS的引用必须以电话号码和调制解调器设置的形式; 用户将不得不挂断，拨打其他BBS，然后手动找到他们正在寻找的信息。 这是不可能的直接链接到另一个BBS内的一些内容。 ↩

按照惯例，许多（但不是全部）Unix程序将这个字节序列视为ASCII文本。我们的日志分析示例使用了这个事实：awk，sort，uniq和head都将它们的输入文件视为由 `\n` （换行符，ASCII 0x0A）字符分隔的记录列表。 `\n` 的选择是任意的 —— 可以说，ASCII记录分隔

符 `0x1E` 本来就是一个更好的选择，因为它是为了这个目的而设计的【14】，但是无论如何，所有这些程序都使用相同的记录分隔符允许它们互操作。

每个记录（即一行输入）的解析更加模糊。 Unix工具通常通过空白或制表符将行分割成字段，但也使用CSV（逗号分隔），管道分隔和其他编码。即使像 `xargs` 这样一个相当简单的工具也有六个命令行选项，用于指定如何解析输入。

ASCII文本的统一接口主要工作，但它不是很漂亮：我们的日志分析示例使用 `{print $7}` 来提取网址，这样可读性不是很好。在理想的世界中，这可能是 `{print $request_url}` 或类似的东西。我们稍后会回到这个想法。

尽管几十年后还不够完美，但统一的Unix接口仍然是非常出色的设计。与Unix工具一样，软件的交互操作和编写并不是很多，你不能通过自定义分析工具轻松地将电子邮件帐户的内容和在线购物历史记录传送到电子表格中，并将结果发布到社交网络或维基。今天，像Unix工具一样流畅地运行程序是一个例外，而不是规范。

即使是具有相同数据模型（**same data model**）的数据库，将数据从一种导到另一种也并不容易。缺乏整合导致了数据的巴尔干化[译注i]。

> 译注i. **巴尔干化（**Balkanization**）是一个常带有贬义的地缘政治学术语，其定义为：一个国家或政区分裂成多个互相敌对的国家或政区的过程。 ↩

逻辑和布线相分离

Unix工具的另一个特点是使用标准输入（stdin）和标准输出（stdout）。如果你运行一个程序，而不指定任何其他的东西，标准输入来自键盘，标准输出指向屏幕。但是，你也可以从文件输入和/或将输出重定向到文件。管道允许你将一个进程的标准输出附加到另一个进程的标准输入（具有小内存缓冲区，而不需要将整个中间数据流写入磁盘）。

程序仍然可以直接读取和写入文件，但如果程序不担心特定的文件路径，只使用标准输入和标准输出，则Unix方法效果最好。这允许shell用户以任何他们想要的方式连接输入和输出;该程序不知道或不关心输入来自哪里以及输出到哪里。 （人们可以说这是一种松耦合（**loose coupling**），晚期绑定（**late binding**）【15】或控制反转（**inversion of control**）【16】）。将输入/输出布线与程序逻辑分开，可以将小工具组合成更大的系统。

你甚至可以编写自己的程序，并将它们与操作系统提供的工具组合在一起。你的程序只需要从标准输入读取输入，并将输出写入标准输出，并且可以参与数据处理流水线。在日志分析示例中，你可以编写一个工具将用户代理字符串转换为更灵敏的浏览器标识符，或者将IP地址转换为国家代码的工具，并将其插入管道。排序程序并不关心它是否与操作系统的另一部分或者你写的程序通信。

但是，使用stdin和stdout可以做什么是有限的。需要多个输入或输出的程序是可能的但棘手的。如果程序直接打开文件进行读取和写入，或者将另一个程序作为子进程启动，或者打开网络连接，则无法将程序的输出传输到网络连接中【17,18】[iii] 。I/O由程序本身连接。它仍然可以配置（例如通过命令行选项），但是减少了在Shell中连接输入和输出的灵活性。

iii

> iii. 除了使用一个单独的工具，如`netcat`或`curl`。 Unix开始试图将所有东西都表示为文件，但是BSD套接字API偏离了这个惯例【17】。研究用操作系统Plan 9和Inferno在使用文件方面更加一致：它们将TCP连接表示为`/net/tcp`中的文件【18】。 ↩

透明度和实验

使Unix工具如此成功的部分原因是它们使得查看正在发生的事情变得非常容易：

- Unix命令的输入文件通常被视为不可变的。这意味着你可以随意运行命令，尝试各种命令行选项，而不会损坏输入文件。
- 你可以在任何时候结束管道，将管道输出到`less`，然后查看它是否具有预期的形式。这种检查能力对调试非常有用。
- 你可以将一个流水线阶段的输出写入文件，并将该文件用作下一阶段的输入。这使你可以重新启动后面的阶段，而无需重新运行整个管道。

因此，与关系数据库的查询优化器相比，即使Unix工具非常简单，工具简单，但仍然非常有用，特别是对于实验而言。

然而，Unix工具的最大局限在于它们只能在一台机器上运行 —— 而Hadoop这样的工具即为此而生。

MapReduce和分布式文件系统

MapReduce有点像Unix工具，但分布在数千台机器上。像Unix工具一样，这是一个相当直接的，蛮力的，但却是令人惊讶的有效工具。一个MapReduce作业可以和一个Unix进程相媲美：它需要一个或多个输入，并产生一个或多个输出。

和大多数Unix工具一样，运行MapReduce作业通常不会修改输入，除了生成输出外没有任何副作用。输出文件以连续的方式写入一次（一旦写入文件，不会修改任何现有的文件部分）。

虽然Unix工具使用stdin和stdout作为输入和输出，但MapReduce作业在分布式文件系统上读写文件。在Hadoop的Map-Reduce实现中，该文件系统被称为HDFS（Hadoop分布式文件系统），一个开源的重新实现Google文件系统（GFS）【19】。

除HDFS外，还有各种其他分布式文件系统，如GlusterFS和Quantcast File System（QFS）【20】。诸如Amazon S3，Azure Blob存储和OpenStack Swift 【21】等对象存储服务在很多方面都是相似的[iv]。在本章中，我们将主要使用HDFS作为示例，但是这些原则适用于任何分布式文件系统。

iv

> [iv]. 一个不同之处在于，对于HDFS，可以将计算任务安排在存储特定文件副本的计算机上运行，而对象存储通常将存储和计算分开。如果网络带宽是一个瓶颈，从本地磁盘读取有性能优势。但是请注意，如果使用删除编码，局部优势将会丢失，因为来自多台机器的数据必须进行合并以重建原始文件【20】。↩

与网络连接存储（NAS）和存储区域网络（SAN）架构的共享磁盘方法相比，HDFS基于无共享原则（参见第二部分的介绍）。共享磁盘存储由集中式存储设备实现，通常使用定制硬件和专用网络基础设施（如光纤通道）。另一方面，无共享方法不需要特殊的硬件，只需要通过传统数据中心网络连接的计算机。

HDFS包含在每台机器上运行的守护进程，暴露一个允许其他节点访问存储在该机器上的文件的网络服务（假设数据中心中的每台通用计算机都附带有一些磁盘）。名为NameNode的中央服务器会跟踪哪个文件块存储在哪台机器上。因此，HDFS在概念上创建一个可以使用运行守护进程的所有机器的磁盘上的空间的大文件系统。

为了容忍机器和磁盘故障，文件块被复制到多台机器上。复制可能意味着多个机器上的相同数据的多个副本，如第5章中所述，或者像Reed-Solomon代码这样的擦除编码方案，它允许以比完全复制更低的存储开销恢复丢失的数据【20,22】。这些技术与RAID相似，可以在连接到同一台机器的多个磁盘上提供冗余;区别在于在分布式文件系统中，文件访问和复制是在传统的数据中心网络上完成的，没有特殊的硬件。

HDFS已经很好地扩展了：在撰写本文时，最大的HDFS部署运行在成千上万台机器上，总存储容量达数百PB【23】。如此大的规模已经变得可行，因为使用商品硬件和开源软件的HDFS上的数据存储和访问成本远低于专用存储设备上的同等容量【24】。

MapReduce作业执行

MapReduce是一个编程框架，你可以使用它编写代码来处理HDFS等分布式文件系统中的大型数据集。理解它的最简单方法是参考“简单日志分析”中的Web服务器日志分析示例。MapReduce中的数据处理模式与此示例非常相似：

1. 读取一组输入文件，并将其分解成记录。在Web服务器日志示例中，每条记录都是日志中的一行（即 `\n` 是记录分隔符）。
2. 调用Mapper函数从每个输入记录中提取一个键和值。在前面的例子中，mapper函数是 `awk'{print $7}'` ：它提取 `URL($7)` 作为关键字，并将值保留为空。
3. 按键排序所有的键值对。在日志示例中，这由第一个排序命令完成。
4. 调用reducer函数遍历排序后的键值对。如果同一个键出现多次，排序使它们在列表中相邻，所以很容易组合这些值而不必在内存中保留很多状态。在前面的例子中，reducer是由 `uniq -c` 命令实现的，该命令使用相同的键来统计相邻记录的数量。

这四个步骤可以由一个MapReduce作业执行。步骤2（Map）和4（Reduce）是你编写自定义数据处理代码的地方。步骤1（将文件分解成记录）由输入格式解析器处理。步骤3中的排序步骤隐含在MapReduce中 —— 你不必编写它，因为Mapper的输出始终在送往reducer之前进

行排序。

要创建MapReduce作业，你需要实现两个回调函数，mapper和reducer，其行为如下（参阅“MapReduce查询”）：

Mapper

每个输入记录都会调用一次Mapper，其工作是从输入记录中提取键和值。对于每个输入，它可以生成任意数量的键值对（包括None）。它不会保留从一个输入记录到下一个记录的任何状态，因此每个记录都是独立处理的。

Reducer MapReduce框架采用由Mapper生成的键值对，收集属于同一个键的所有值，并使用迭代器调用reducer以调用该值集合。 Reducer可以产生输出记录（例如相同URL的出现次数）。

在Web服务器日志示例中，我们在第5步中有第二个排序命令，它按请求数对URL进行排序。在MapReduce中，如果你需要第二个排序阶段，则可以通过编写第二个MapReduce作业并将第一个作业的输出用作第二个作业的输入来实现它。这样看来，Mapper的作用是将数据放入一个适合排序的表单中，并且Reducer的作用是处理已排序的数据。

分布式执行MapReduce

Unix命令管道的主要区别在于，MapReduce可以在多台机器上并行执行计算，而无需编写代码来明确处理并行性。Mapper和Reducer一次只能处理一条记录;他们不需要知道他们的输入来自哪里或者输出什么地方，所以框架可以处理机器之间移动数据的复杂性。

在分布式计算中可以使用标准的Unix工具作为Mapper和Reducer【25】，但更常见的是，它们被实现为传统编程语言的函数。在Hadoop MapReduce中，Mapper和Reducer都是实现特定接口的Java类。在MongoDB和CouchDB中，Mapper和Reducer都是JavaScript函数（参阅“MapReduce查询”）。

图10-1显示了Hadoop MapReduce作业中的数据流。其并行化基于分区（参见第6章）：作业的输入通常是HDFS中的一个目录，输入目录中的每个文件或文件块都被认为是一个单独的分区，可以单独处理map任务（图10-1中的m1，m2和m3标记）。

每个输入文件的大小通常是数百兆字节。 MapReduce调度器（图中未显示）试图在其中一台存储输入文件副本的机器上运行每个Mapper，只要该机器有足够的备用RAM和CPU资源来运行映射任务【26】。这个原则被称为将数据放在数据附近【27】：它节省了通过网络复制输入文件，减少网络负载和增加局部性。

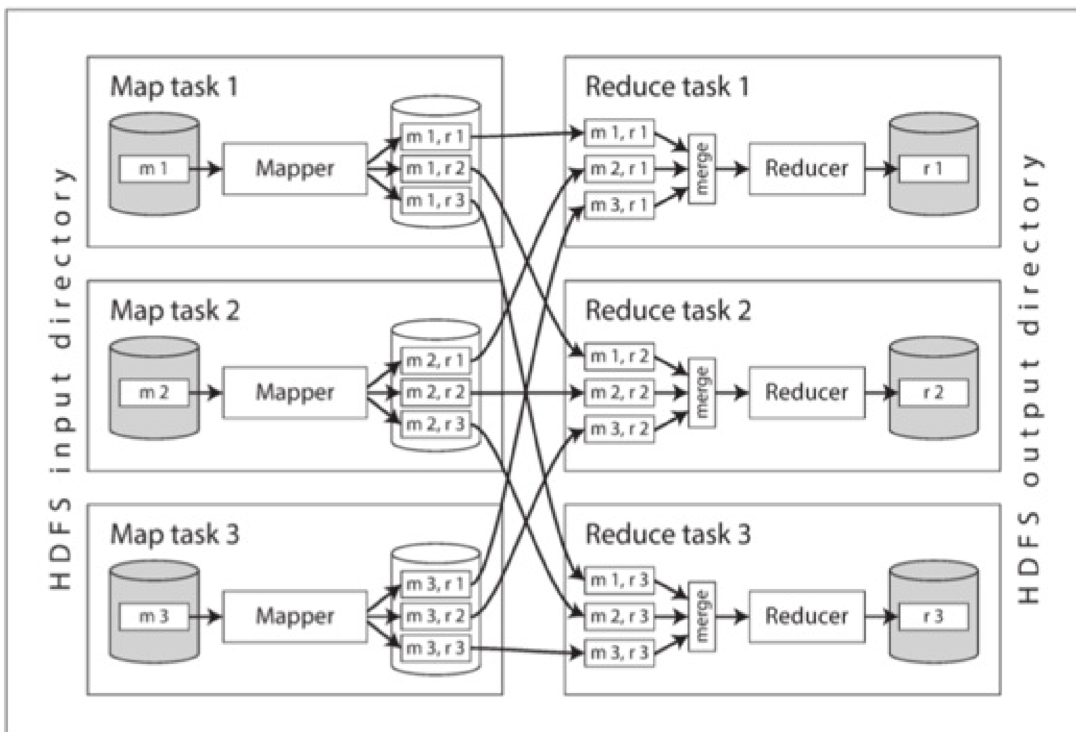


图**10-1** 具有三个**Mapper**和三个**Reducer**的**MapReduce**任务

在大多数情况下，应该在映射任务中运行的应用程序代码在分配运行它的任务的计算机上还不存在，所以MapReduce框架首先复制代码（例如Java程序中的JAR文件）到适当的机器。然后启动Map任务并开始读取输入文件，一次将一条记录传递给Mapper回调。Mapper的输出由键值对组成。

计算的减少方面也被分割。虽然Map任务的数量由输入文件块的数量决定，但Reducer的任务的数量是由作业作者配置的（它可以不同于Map任务的数量）。为了确保具有相同键的所有键值对在相同的Reducer处结束，框架使用键的散列值来确定哪个减少的任务应该接收到特定的键值对（参见“按键散列分区”））。

键值对必须进行排序，但数据集可能太大，无法在单台机器上使用常规排序算法进行排序。相反，分类是分阶段进行的。首先，每个映射任务都基于键的散列，通过Reducer分割其输出。这些分区中的每一个都被写入映射程序本地磁盘上的已排序文件，使用的技术与我们在“SSTables与LSM树”中讨论的类似。

只要Mapper完成读取输入文件并写入其排序后的输出文件，MapReduce调度器就会通知减速器他们可以从该Mapper开始获取输出文件。减法器连接到每个Mapper，并为其分区下载排序后的键值对的文件。通过简化，分类和将数据分区从Mapper复制到Reducer的过程被称为混洗（**shuffle**）【26】（一个令人困惑的术语 —— 不像洗牌一样，在MapReduce中没有随机性）。

reduce任务从Mapper获取文件并将它们合并在一起，并保存排序顺序。因此，如果不同的Mapper使用相同的键生成记录，则它们将在合并的Reducer输入中相邻。

使用一个键和一个迭代器调用reducer，迭代器使用相同的键（在某些情况下可能不是全部适合内存）逐步扫描所有记录。 Reducer可以使用任意逻辑来处理这些记录，并且可以生成任意数量的输出记录。这些输出记录写入到分布式文件系统上的文件（通常是运行reducer的机器的本地磁盘上的一个副本，其他机器上的副本）。

MapReduce工作流

单个MapReduce作业可以解决的问题范围有限。参阅日志分析示例，一个MapReduce作业可以确定每个URL的页面浏览次数，但不是最常用的URL，因为这需要第二轮排序。

因此，将MapReduce作业链接到工作流中是非常常见的，例如，一个作业的输出成为下一个作业的输入。 Hadoop Map-Reduce框架对工作流程没有特别的支持，所以这个链接是通过目录名隐含完成的：第一个作业必须被配置为将其输出写入HDFS中的指定目录，第二个作业必须是配置为读取与其输入相同的目录名称。从MapReduce框架的角度来看，他们是两个独立的工作。

因此，被链接的MapReduce作业不如Unix命令的流水线（它直接将一个进程的输出作为输入传递给另一个进程，只使用一个小的内存缓冲区），更像是一系列命令，其中每个命令的输出写入临时文件，下一个命令从临时文件中读取。这种设计有利有弊，我们将在“物化中间状态”中讨论。

当作业成功完成时，批处理作业的输出仅被视为有效（MapReduce丢弃失败作业的部分输出）。因此，工作流程中的一项工作只有在先前的工作 —— 即生产其投入方向的工作 —— 成功完成时才能开始。处理这些作业之间的依赖关系执行，为Hadoop开发了各种工作流调度器，包括Oozie，Azkaban，Luigi，Airflow和Pinball 【28】。

这些调度程序还具有管理功能，在维护大量批处理作业时非常有用。在构建推荐系统【29】时，由50到100个MapReduce作业组成的工作流是常见的，而在大型组织中，许多不同的团队可能运行不同的作业来读取彼此的输出。工具支持对于管理这样复杂的数据流非常重要。

Hadoop的各种高级工具（如Pig 【30】，Hive 【31】，Cascading 【32】，Crunch 【33】和FlumeJava 【34】）也设置了多个MapReduce阶段的工作流程 。

Reduce端连接与分组

我们在第2章中讨论了数据模型和查询语言的联接，但是我们还没有深入探讨连接是如何实现的。现在是我们再次拿起那个线程的时候了。

在许多数据集中，通常一条记录与另一条记录有关联：关系模型中的外键，文档模型中的文档引用或图模型中的边。只要有一些代码需要访问该关联两边的记录（包含引用的记录和被引用的记录），连接就是必需的。正如第2章所讨论的，非规范化可以减少对连接的需求，但通常不会将其完全移除[V]。

在数据库中，如果执行只涉及少量记录的查询，数据库通常会使用索引来快速定位感兴趣的记录（参阅第3章）。如果查询涉及连接，则可能需要多个索引查找。然而，MapReduce没有索引的概念 —— 至少不是通常意义上的。

当MapReduce作业被赋予一组文件作为输入时，它读取所有这些文件的全部内容；一个数据库会调用这个操作一个全表扫描。如果你只想读取少量的记录，则与索引查找相比，全表扫描的成本非常高昂。但是，在分析查询中（参阅"事务处理或分析？"），通常需要计算大量记录的聚合。在这种情况下，扫描整个输入可能是相当合理的事情，特别是如果可以在多台机器上并行处理。

> v. 我们在本书中讨论的连接通常是等值连接，即最常见的连接类型，其中记录与其他记录在特定字段（例如ID）中具有相同的值相关联。有些数据库支持更一般的连接类型，例如使用小于运算符而不是等号运算符，但是我们没有空间来覆盖它们。↩

当我们在批处理的背景下讨论连接时，我们的意思是解决数据集内某个关联的所有事件。 例如，我们假设一个工作是同时为所有用户处理数据，而不仅仅是为一个特定用户查找数据（这可以通过索引更有效地完成）。

示例：分析用户活动事件

图10-2给出了一个批处理作业中加入典型的例子。 在左侧是事件日志，描述登录用户在网站上做的事情（称为活动事件或点击流数据），右侧是用户数据库。 你可以将此示例看作是星型模式的一部分（参阅"星型和雪花型：分析的模式"）：事件日志是事实表，用户数据库是其中一个纬度。

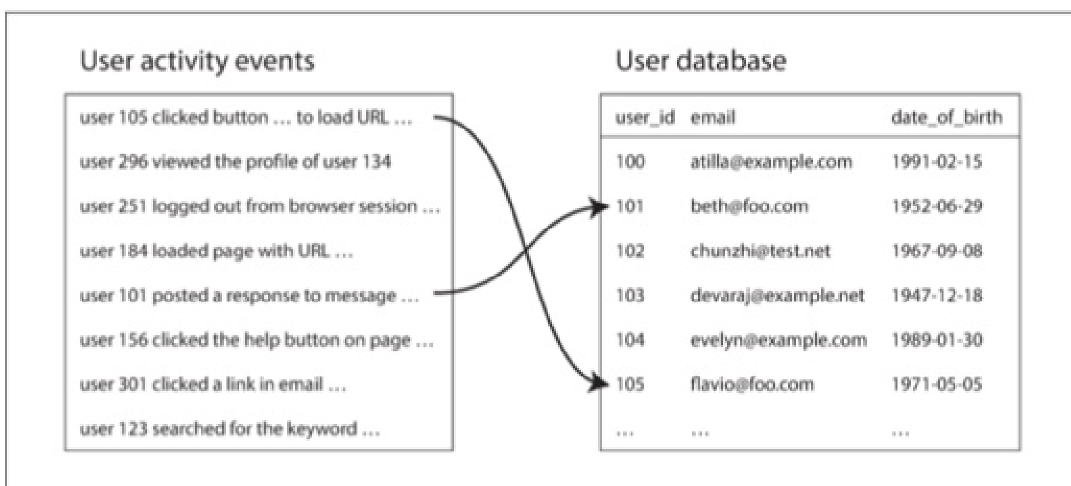


图**10-2** 用户行为日志与用户档案的连接

分析任务可能需要将用户活动与用户简档信息相关联：例如，如果简档包含用户的年龄或出生日期，则系统可以确定哪些年龄组最受欢迎。但是，活动事件仅包含用户标识，而不包含完整的用户配置文件信息。在每一个活动事件中嵌入这个简介信息很可能是非常浪费的。因此，活动事件需要加入用户配置文件数据库。

这个连接的最简单实现将逐个遍历活动事件，并为每个遇到的用户ID查询用户数据库（在远程服务器上）。这是可能的，但是它很可能会遭受非常差的性能：处理吞吐量将受到数据库服务器的往返时间的限制，本地缓存的有效性将很大程度上取决于数据的分布，并行运行大量查询可能会轻易压倒数据库【35】。

为了在批处理过程中实现良好的吞吐量，计算必须（尽可能）在一台机器上进行。通过网络为你要处理的每个记录进行随机访问请求太慢。而且，查询远程数据库意味着批处理作业变得不确定，因为远程数据库中的数据可能会改变。

因此，更好的方法是获取用户数据库的副本（例如，使用ETL进程从数据库备份中提取数据，参阅“数据仓库”），并将其放入与日志相同的分布式文件系统用户活动事件。然后，你可以将用户数据库存储在HDFS中的一组文件中，并将用户活动记录在另一组文件中，并且可以使用MapReduce将所有相关记录集中到同一地点并高效地处理它们。

排序合并连接

回想一下，Mapper的目的是从每个输入记录中提取一个键和值。在图10-2的情况下，这个键就是用户ID：一组Mapper会覆盖活动事件（提取用户ID作为键和活动事件作为值），而另一组Mapper将会检查用户数据库（提取用户ID作为键和用户的出生日期作为值）。这个过程如图10-3所示。

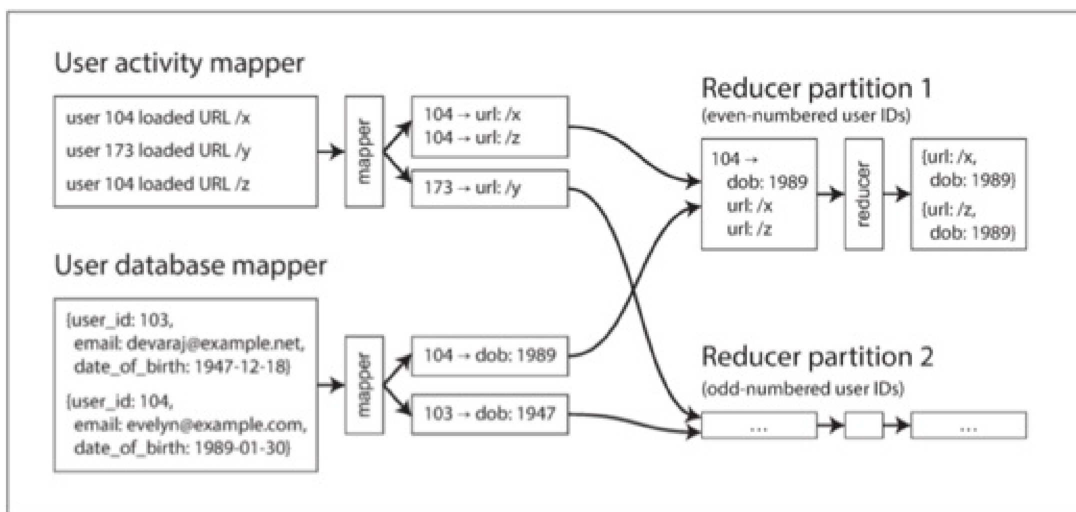


图**10-3 Reduce**端在**user ID**上进行归并排序连接，如果输入数据集分片成多个文件，则每个都会被多个**Mapper**并行处理

当MapReduce框架通过key对mapper输出进行分区，然后对键值对进行排序时，效果是所有活动事件和用户ID相同的用户记录在reducer输入中彼此相邻。 Map-Reduce作业甚至可以安排记录进行排序，使减速器始终如一 首先从用户数据库中查看记录，然后按照时间戳顺序查看活动事件 —— 这种技术被称为次级排序【26】。

然后reducer可以很容易地执行实际的加入逻辑：每个用户ID调用一次reducer函数，并且由于二次排序，第一个值应该是来自用户数据库的出生日期记录。 Reducer将出生日期存储在局部变量中，然后使用相同的用户ID遍历活动事件，输出已观看网址和观看者年龄对。随后的

Map-Reduce作业可以计算每个URL的查看者年龄分布，并按年龄组进行聚类。

由于reducer一次处理一个特定用户ID的所有记录，因此只需要一次将一个用户记录保存在内存中，而不需要通过网络发出任何请求。这个算法被称为排序合并连接，因为Mapper输出是按键排序的，然后Reducer将来自连接两边的排序的记录列表合并在一起。

把相关数据放在一起

在排序合并连接中，Mapper和排序过程确保将执行特定用户标识的连接操作的所有必需数据放在一起：一次调用reducer。预先排列了所有需要的数据，reducer可以是一个相当简单，单线程的代码，可以通过高吞吐量和低内存开销通过记录。

查看这种体系结构的一种方法是Mapper将“消息”发送给reducer。当一个Mapper发出一个键值对时，这个键的作用就像值应该传递到的目标地址。即使键只是一个任意的字符串（不是像IP地址和端口号那样的实际的网络地址），它的行为就像一个地址：所有具有相同键的键对将被传送到相同的目标（一次Reduce的调用）。

使用MapReduce编程模型将计算的物理网络通信方面（从正确的计算机获取数据）从应用程序逻辑中分离出来（处理完数据后）。这种分离与数据库的典型使用形成了鲜明的对比，从数据库中获取数据的请求经常发生在应用程序代码的深处【36】。由于MapReduce能够处理所有的网络通信，因此它也避免了应用程序代码担心部分故障，例如另一个节点的崩溃：MapReduce在不影响应用程序逻辑的情况下，透明地重试失败的任务。

GROUP BY

除了连接之外，“将相关数据引入同一地点”模式的另一个常见用法是通过某个键（如SQL中的GROUP BY子句）对记录进行分组。所有用相同的键记录一个组，并且下一步往往是在每个组内进行某种聚合，例如：

- 计算每个组中记录的数量（例如，在统计页面视图的示例中，你将在SQL中表示为COUNT（*）聚合）
- 在SQL中的一个特定字段（ `SUM(fieldname)` ）中添加值
- 根据某些排名函数选择前k个记录

使用MapReduce实现这种分组操作的最简单方法是设置Mapper，以便它们生成的键值对使用所需的分组键。然后，分区和排序过程将所有记录与同一个Reducer中的相同键集合在一起。因此，在MapReduce上实现时，分组和连接看起来非常相似。

分组的另一个常见用途是整理特定用户会话的所有活动事件，以便找出用户采取的一系列操作（称为会话化【37】）。例如，可以使用这种分析来确定显示网站新版本的用户是否比那些显示旧版本（A/B测试）的用户更有可能进行购买，或计算某个营销活动是值得的。

如果你有多个Web服务器处理用户请求，则特定用户的活动事件很可能分散在各个不同的服务器的日志文件中。你可以通过使用会话cookie，用户ID或类似的标识符作为分组键来实现会话，并将特定用户的所有活动事件放在一起，同时将不同用户的事件分配到不同的分区。

处理倾斜

如果存在与单个键相关的大量数据，则“将具有相同键的所有记录带到相同位置”的模式被破坏。例如，在社交网络中，大多数用户可能会连接到几百人，但少数名人可能有数百万的追随者。这种不成比例的活动数据库记录被称为关键对象【38】或热键。

在单个Reducer中收集与名人相关的所有活动（例如回复他们发布的内容）可能导致严重的倾斜（也称为热点）—— 也就是说，一个减速器必须处理比其他更多的记录（参见“负载倾斜与消除热点“）。由于MapReduce作业只有在其所有Mapper和Reducer都完成时才完成，所有后续作业必须等待最慢的Reducer才能启动。

如果加入输入有热点键，则可以使用一些算法进行补偿。例如，Pig中的倾斜连接方法首先运行一个抽样作业来确定哪些键是热的【39】。执行实际加入时，Mapper发送任何随机选择一个与几个减速器之一相关的热键的记录（与传统的MapReduce相比，它选择一个基于键散列确定性的减速器）。对于加入的其他输入，与热键相关的记录需要被复制到所有处理该键的Reducer【40】。

这种技术将处理热键的工作分散到多个reducer上，这样可以使其更好地并行化，而不必将其他join连接复制到多个reducer。 Crunch中的分片连接方法是相似的，但需要显式指定热键而不是使用采样作业。这种技术也非常类似于我们在“负载倾斜与消除热点”中讨论的技术，使用随机化来缓解分区数据库中的热点。

Hive的偏斜连接优化采取了另一种方法。它需要在表格元数据中明确指定热键，并将与这些键相关的记录与其余文件分开存放。在该表上执行连接时，它将使用Map边连接（参阅下一节）获取热键。

使用热键对记录进行分组并汇总记录时，可以分两个阶段进行分组。第一个MapReduce阶段将记录发送到随机Reducer，以便每个Reducer对热键的记录子集执行分组，并为每个键输出更紧凑的聚合值。第二个Map-Reduce作业然后将来自所有第一阶段减速器的值合并为每个键的单个值。

Map端连接

上一节描述的连接算法在reducer中执行实际的连接逻辑，因此被称为reduce-side连接。Mapper扮演着输入数据的角色：从每个输入记录中提取键和值，将键值对分配给reducer分区，并按键排序。

减少方法的优点是不需要对输入数据做任何假设：无论其属性和结构如何，Mapper都可以准备数据以准备加入。然而，不利的一面是，所有这些排序，复制到Reducer以及合并减速器输入可能是非常昂贵的。取决于可用的内存缓冲区，当数据通过MapReduce 【37】阶段时，数

据可能被写入磁盘几次。

另一方面，如果你可以对输入数据进行某些假设，则可以通过使用所谓的map端连接来加快连接速度。这种方法使用了一个Reduce的MapReduce作业，其中没有减速器，也没有排序。相反，每个Mapper只需从分布式文件系统读取一个输入文件块，然后将一个输出文件写入文件系统即可。

广播散列连接

执行Map端连接最简单的方法适用于大数据集与小数据集连接的情况。特别是，小数据集需要足够小，以便可以将其全部加载到每个Mapper的内存中。

例如，假设在图10-2的情况下，用户数据库足够小以适应内存。在这种情况下，当Mapper启动时，它可以首先将用户数据库从分布式文件系统读取到内存中的哈希表中。完成此操作后，Map程序可以扫描用户活动事件，并简单地查找散列表中每个事件的用户标识[vi]。

> [vi]. 这个例子假定散列表中的每个键只有一个条目，这对用户数据库（用户ID唯一标识一个用户）可能是正确的。通常，哈希表可能需要包含具有相同键的多个条目，并且连接运算符将输出关键字的所有匹配。↩

仍然可以有几个映射任务：一个用于连接的大输入的每个文件块（在图10-2的例子中，活动事件是大输入）。这些Mapper中的每一个都将小输入全部加载到内存中。

这种简单而有效的算法被称为广播散列连接：广播一词反映了这样一个事实，即大输入的分区的每个Mapper都读取整个小输入（所以小输入有效地“广播”到大的输入），单词hash反映了它使用一个哈希表。 Pig（名为“replicated join”），Hive（“MapJoin”），Cascading和Crunch支持此连接方法。它也用于数据仓库查询引擎，如Impala【41】。

而不是将小连接输入加载到内存散列表中，另一种方法是将小连接输入存储在本地磁盘上的只读索引中【42】。该索引中经常使用的部分将保留在操作系统的页面缓存中，因此这种方法可以提供与内存中哈希表几乎一样快的随机访问查找，但实际上并不需要数据集适合内存。

分区散列连接

如果以相同方式对映射端连接的输入进行分区，则散列连接方法可以独立应用于每个分区。在图10-2的情况下，你可以根据用户标识的最后一位十进制数字来安排活动事件和用户数据库的每一个（因此每边有10个分区）。例如，Mapper3首先将所有具有以3结尾的ID的用户加载到散列表中，然后扫描ID为3的每个用户的所有活动事件。

如果分区正确完成，你可以确定所有你可能要加入的记录都位于相同编号的分区中，因此每个Mapper只能从每个输入数据集中读取一个分区就足够了。这具有的优点是每个Mapper都可以将较少量的数据加载到其哈希表中。

这种方法只适用于两个连接的输入具有相同数量的分区，记录根据相同的键和相同的散列函数分配给分区。如果输入是由之前执行过这个分组的MapReduce作业生成的，那么这可能是一个合理的假设。

分区散列连接在Hive 【37】中称为bucketed映射连接。Map端合并连接

如果输入数据集不仅以相同的方式进行分区，而且还基于相同的键进行排序，则应用另一种Map端联接的变体。在这种情况下，输入是否足够小以适应内存并不重要，因为Mapper可以执行通常由reducer执行的相同合并操作：按递增键递增读取两个输入文件，以及匹配相同的键记录。

如果Map端合并连接是可能的，则可能意味着先前的MapReduce作业首先将输入数据集引入到这个分区和排序的表单中。原则上，这个加入可以在之前工作的Reduce阶段进行。但是，在单独的仅用于Map的作业中执行合并连接仍然是适当的，例如，除了此特定连接之外，还需要分区和排序数据集以用于其他目的。

MapReduce与Map端连接的工作流程

当下游作业使用MapReduce连接的输出时，map-side或reduce-side连接的选择会影响输出的结构。 reduce-side连接的输出按连接键进行分区和排序，而map-side连接的输出按照与大输入相同的方式进行分区和排序（因为对每个文件块启动一个map任务无论是使用分区连接还是广播连接，连接的大输入）。

如前所述，Map边连接也对输入数据集的大小，排序和分区做出了更多的假设。在优化连接策略时，了解分布式文件系统中数据集的物理布局变得非常重要：仅仅知道编码格式和数据存储目录的名称是不够的;你还必须知道数据分区和排序的分区数量和键。

在Hadoop生态系统中，这种关于数据集分区的元数据经常在HCatalog和Hive Metastore中维护【37】。

工作流的输出

我们已经谈了很多关于实现MapReduce工作流程的各种算法，但是我们忽略了一个重要的问题：一旦完成，所有处理的结果是什么？我们为什么要把所有这些工作放在首位？

在数据库查询的情况下，我们根据分析目的来区分事务处理（OLTP）目的（参阅“事务处理或分析？”）。我们看到，OLTP查询通常使用索引按键查找少量记录，以便将其呈现给用户（例如，在网页上）。另一方面，分析查询通常会扫描大量记录，执行分组和汇总，输出通常具有报告的形式：显示某个指标随时间变化的图表，或前10个项目根据一些排名，或一些数量分解成子类别。这种报告的消费者通常是需要做出商业决策的分析师或经理。

批处理在哪里适合？这不是交易处理，也不是分析。与分析更接近，因为批处理过程通常扫描输入数据集的大部分。但是，MapReduce作业的工作流程与用于分析目的的SQL查询不同（参阅“比较Hadoop与分布式数据库”）。批处理过程的输出通常不是报告，而是一些其他类

型的结构。

建立搜索索引

Google最初使用的MapReduce是为其搜索引擎建立索引，这个索引是作为5到10个MapReduce作业的工作流实现的【1】。虽然Google为了这个目的后来不再使用MapReduce【43】，但是如果从建立搜索索引的角度来看，它可以帮助理解MapReduce。（即使在今天，Hadoop MapReduce仍然是构建Lucene / Solr索引的好方法。）

我们在“全文搜索和模糊索引”中简要地看到了Lucene这样的全文搜索索引是如何工作的：它是一个文件（关键词字典），你可以在其中高效地查找特定关键字并找到包含该关键字的所有文档ID列表（发布列表）。这是一个非常简单的搜索索引视图 —— 实际上，它需要各种附加数据，以便根据相关性对搜索结果进行排名，纠正拼写错误，解析同义词等等，但这一原则是成立的。

如果需要对一组固定文档执行全文搜索，则批处理是构建索引的一种非常有效的方法：Mapper根据需要对文档集进行分区，每个reducer构建其分区的索引，并将索引文件写入分布式文件系统。构建这样的文档分区索引（参阅“分区和二级索引”）并行处理非常好。

由于按关键字查询搜索索引是只读操作，因此这些索引文件一旦创建就是不可变的。

如果索引的文档集合发生更改，则可以选择定期重新运行整个索引工作流程，并在完成后用新的索引文件批量替换以前的索引文件。如果只有少量的文档发生了变化，这种方法可能会带来很高的计算成本，但是它的优点是索引过程很容易推理：文档，索引。

或者，可以逐渐建立索引。如第3章所述，如果要添加，删除或更新索引中的文档，Lucene会写出新的段文件，并异步合并和压缩背景中的段文件。我们将在第11章中看到更多这样的增量处理。

键值存储作为批处理输出

搜索索引只是批处理工作流程可能输出的一个示例。批量处理的另一个常见用途是构建机器学习系统，如分类器（例如，垃圾邮件过滤器，异常检测，图像识别）和推荐系统（例如，你可能认识的人，你可能感兴趣的产品或相关搜索）。

这些批处理作业的输出通常是某种数据库：例如，可以通过用户ID查询以获取该用户的建议朋友的数据库，或者可以通过产品ID查询的数据库以获取相关产品【45】。

这些数据库需要从处理用户请求的Web应用程序中查询，这些请求通常与Hadoop基础架构分离。那么批处理过程的输出如何返回到Web应用程序可以查询的数据库？

最明显的选择可能是直接在Mapper或Reducer中使用客户端库作为你最喜欢的数据库，并从批处理作业直接写入数据库服务器，一次写入一条记录。这将起作用（假设你的防火墙规则允许从你的Hadoop环境直接访问你的生产数据库），但由于以下几个原因，这是一个坏主意：

- 正如前面讨论的连接一样，为每个记录提出一个网络请求比批处理任务的正常吞吐量要慢几个数量级。即使客户端库支持批处理，性能也可能很差。
- MapReduce作业经常并行运行许多任务。如果所有Mapper或Reducer都同时写入相同的输出数据库，并且批处理过程期望的速率，那么该数据库可能很容易被压倒，并且其查询性能可能受到影响。这可能会导致系统其他部分的操作问题【35】。
- 通常情况下，MapReduce为作业输出提供了一个干净的“全有或全无”的保证：如果作业成功，则结果就是只执行一次任务的输出，即使某些任务失败并且必须重试。如果整个作业失败，则不会生成输出。然而，从作业内部写入外部系统会产生外部可见的副作用，这种副作用是不能被隐藏的。因此，你不得不担心部分完成的作业对其他系统可见的结果，以及Hadoop任务尝试和推测性执行的复杂性。

更好的解决方案是在批处理作业中创建一个全新的数据库，并将其作为文件写入分布式文件系统中作业的输出目录，就像上一节的搜索索引一样。这些数据文件一旦写入就是不可变的，可以批量加载到处理只读查询的服务器中。各种键值存储支持在MapReduce作业中构建数据库文件，包括Voldemort 【46】，Terrapin 【47】，ElephantDB 【48】和HBase批量加载【49】。

构建这些数据库文件是MapReduce的一个很好的使用方法：使用Mapper提取一个键，然后使用该键进行排序已经成为构建索引所需的大量工作。由于大多数这些键值存储是只读的（文件只能由批处理作业一次写入，而且是不可变的），所以数据结构非常简单。例如，它们不需要WAL（参阅“使B树可靠”）。

将数据加载到Voldemort时，服务器将继续向旧数据文件提供请求，同时将新数据文件从分布式文件系统复制到服务器的本地磁盘。一旦复制完成，服务器会自动切换到查询新文件。如果在这个过程中出现任何问题，它可以很容易地再次切换回旧的文件，因为它们仍然存在，并且是不变的【46】。

批量过程输出的哲学

本章前面讨论过的Unix哲学（“Unix哲学”）鼓励通过对数据流的非常明确的实验来进行实验：程序读取输入并写入输出。在这个过程中，输入保持不变，任何以前的输出都被新输出完全替换，并且没有其他副作用。这意味着你可以随心所欲地重新运行一个命令，调整或调试它，而不会扰乱系统的状态。

MapReduce作业的输出处理遵循相同的原理。通过将输入视为不可变且避免副作用（如写入外部数据库），批处理作业不仅实现了良好的性能，而且更容易维护：

- 如果在代码中引入了一个错误，并且输出错误或损坏了，则可以简单地回滚到代码的先前版本，然后重新运行该作业，输出将再次正确。或者，甚至更简单，你可以将旧的输出保存在不同的目录中，然后切换回原来的目录。具有读写事务的数据库没有这个属性：如果你部署了错误的代码，将错误的数据写入数据库，那么回滚代码将无法修复数据库中的数据。 （能够从错误代码中恢复的思想被称为人类容错【50】。）
- 由于易于回滚，功能开发可以比错误意味着不可挽回的损害的环境更快地进行。这种使

不可逆性最小化的原则有利于敏捷软件的开发【51】。

- 如果映射或减少任务失败，MapReduce框架将自动重新调度并在同一个输入上再次运行它。如果失败是由于代码中的一个错误造成的，那么它会一直崩溃，并最终导致作业在几次尝试之后失败。但是如果故障是由于暂时的问题引起的，那么故障是可以容忍的。这种自动重试只是安全的，因为输入是不可变的，而失败任务的输出被MapReduce框架丢弃。
- 同一组文件可用作各种不同作业的输入，其中包括计算度量标准的计算作业，并评估作业的输出是否具有预期的特性（例如，将其与前一次运行的输出进行比较并测量差异）。
- 与Unix工具类似，MapReduce作业将逻辑与布线（配置输入和输出目录）分开，这就提供了关注点的分离，并且可以重用代码：一个团队可以专注于实现一件好事的工作其他团队可以决定何时何地运行这项工作。

在这些领域，对Unix运行良好的设计原则似乎也适用于Hadoop，但Unix和Hadoop在某些方面也有所不同。例如，因为大多数Unix工具都假定没有类型的文本文件，所以他们必须做大量的输入解析（本章开头的日志分析示例使用 `{print $7}` 来提取URL）。在Hadoop上，通过使用更多结构化的文件格式，可以消除一些低价值的语法转换：Avro（参阅“Avro”）和Parquet（参阅第95页上的“列存储”）经常使用，因为它们提供高效的基于模式的编码，并允许随着时间的推移模式的演变（见第4章）。

比较Hadoop和分布式数据库

正如我们所看到的，Hadoop有点像Unix的分布式版本，其中HDFS是文件系统，而MapReduce是Unix进程的古怪实现（这恰好总是在映射阶段和缩小阶段之间运行排序实用程序）。我们看到了如何在这些基元之上实现各种连接和分组操作。

当MapReduce论文【1】发表时，它在某种意义上说并不新鲜。我们在前几节中讨论的所有处理和并行连接算法已经在十多年前的所谓的大规模并行处理（**MPP， massively parallel processing**）数据库中实现了【3,40】。例如，Gamma数据库机器，Teradata和Tandem NonStop SQL是这方面的先驱【52】。

最大的区别是MPP数据库集中于在一组机器上并行执行分析SQL查询，而MapReduce和分布式文件系统【19】的组合则更像是一个可以运行任意程序的通用操作系统。

存储的多样性

数据库要求你根据特定的模型（例如关系或文档）来构造数据，而分布式文件系统中的文件只是字节序列，可以使用任何数据模型和编码来编写。它们可能是数据库记录的集合，但同样可以是文本，图像，视频，传感器读数，稀疏矩阵，特征向量，基因组序列或任何其他类型的数据。

说白了，Hadoop开放了将数据不加区分地转储到HDFS的可能性，之后才想出如何进一步处理它【53】。相比之下，在将数据导入数据库专有存储格式之前，MPP数据库通常需要对数据和查询模式进行仔细的前期建模。

从纯粹的角度来看，这种仔细的建模和导入似乎是可取的，因为这意味着数据库的用户有更好的质量数据来处理。然而，在实践中，似乎只是简单地使数据可用 —— 即使它是一个古怪的，难以使用的原始格式 —— 通常比尝试决定理想的数据模型更有价值[54]。

这个想法与数据仓库类似（参阅“数据仓库”）：将大型组织的各个部分的数据集中在一起是很有价值的，因为它可以跨以前不同的数据集进行联接。 MPP数据库所要求的谨慎的模式设计减慢了集中式数据收集速度;以原始形式收集数据，以后担心模式设计，使数据收集速度加快（有时被称为“数据湖（**data lake**）”或“企业数据中心（**enterprise data hub**）”【55】）。

不加区别的数据倾销改变了解释数据的负担：不是强迫数据集的生产者将其转化为标准化的格式，而是数据的解释成为消费者的问题（读时模式方法【56】；参阅“文档模型中的架构灵活性”）。如果生产者和消费者是不同优先级的不同团队，这可能是一个优势。甚至可能不存在一个理想的数据模型，而是对适合不同目的的数据有不同的看法。以原始形式简单地转储数据可以进行多次这样的转换。这种方法被称为寿司原则：“原始数据更好”【57】。

因此，Hadoop经常被用于实现ETL过程（参阅“数据仓库”）：事务处理系统中的数据以某种原始形式转储到分布式文件系统中，然后编写MapReduce作业来清理数据，将其转换为关系表单，并将其导入MPP数据仓库以进行分析。数据建模仍然在发生，但它是在一个单独的步骤中，从数据收集中分离出来的。这种解耦是可能的，因为分布式文件系统支持以任何格式编码的数据。

加工模型的多样性

MPP数据库是单一的，紧密集成的软件，负责磁盘上的存储布局，查询计划，调度和执行。由于这些组件都可以针对数据库的特定需求进行调整和优化，因此整个系统可以在其设计的查询类型上取得非常好的性能。而且，SQL查询语言允许表达式查询和重要语义，而无需编写代码，使业务分析师（例如Tableau）使用的图形工具可访问该语言。

另一方面，并非所有类型的处理都可以合理地表达为SQL查询。例如，如果要构建机器学习和推荐系统，或者使用相关性排名模型的全文搜索索引，或者执行图像分析，则很可能需要更一般的数据处理模型。这些类型的处理通常对特定的应用程序非常具体（例如机器学习的特征工程，机器翻译的自然语言模型，欺诈预测的风险评估函数），因此它们不可避免地需要编写代码，而不仅仅是查询。

MapReduce使工程师能够轻松地在大型数据集上运行自己的代码。如果你有HDFS和MapReduce，那么你可以在它上面建立一个SQL查询执行引擎，事实上这正是Hive项目所做的【31】。但是，你也可以编写许多其他形式的批处理，这些批处理不适合用SQL查询表示。

随后，人们发现MapReduce对于某些类型的处理来说太过于限制，执行得太差，因此其他各种处理模型都是在Hadoop之上开发的（我们将在“后MapReduce时代”中看到其中的一些）。有两种处理模型，SQL和MapReduce，还不够，需要更多不同的模型！而且由于Hadoop平台的开放性，实施一整套方法是可行的，而这在整体MPP数据库的范围内是不可能的【58】。

至关重要的是，这些不同的处理模型都可以在一个共享的机器上运行，所有这些机器都可以访问分布式文件系统上的相同文件。在Hadoop方法中，不需要将数据导入到几个不同的专用系统中进行不同类型的处理：系统足够灵活，可以支持同一个群集内不同的工作负载。不需要移动数据使得从数据中获得价值变得容易得多，并且使用新的处理模型更容易进行实验。

Hadoop生态系统包括随机访问的OLTP数据库，如HBase（参阅“SSTables和LSM树”）和MPA样式的分析数据库，如Impala 【41】。 HBase和Impala都不使用MapReduce，但都使用HDFS进行存储。它们是访问和处理数据的非常不同的方法，但是它们可以共存并被集成到同一个系统中。

为频繁的故障而设计

在比较MapReduce和MPP数据库时，设计方法的另外两个不同点是：处理故障和使用内存和磁盘。与在线系统相比，批处理对故障不太敏感，因为如果失败，用户不会立即影响用户，并且可以再次运行。

如果一个节点在执行查询时崩溃，大多数MPP数据库会中止整个查询，并让用户重新提交查询或自动重新运行它【3】。由于查询通常最多运行几秒钟或几分钟，所以这种处理错误的方法是可以接受的，因为重试的代价不是太大。 MPP数据库还倾向于在内存中保留尽可能多的数据（例如，使用散列连接）以避免从磁盘读取的成本。

另一方面，MapReduce可以容忍映射或减少任务的失败，而不会影响作业的整体，通过以单个任务的粒度重试工作。它也非常渴望将数据写入磁盘，一方面是为了容错，另一方面是假设数据集太大而不能适应内存。

MapReduce方法更适用于较大的作业：处理如此之多的数据并运行很长时间的作业，以至于在此过程中可能至少遇到一个任务故障。在这种情况下，由于单个任务失败而重新运行整个工作将是浪费的。即使以单个任务的粒度进行恢复引入了使得无故障处理更慢的开销，但如果任务失败率足够高，仍然可以进行合理的权衡。

但是这些假设有多现实呢？在大多数集群中，机器故障确实发生，但是它们不是很频繁——可能很少，大多数工作都不会经验，因为机器故障。为了容错，真的值得引起重大的开销吗？

要了解MapReduce节省使用内存和任务级恢复的原因，查看最初设计MapReduce的环境是很有帮助的。 Google拥有混合使用的数据中心，在线生产服务和离线批处理作业在同一台机器上运行。每个任务都有一个使用容器执行的资源分配（CPU核心，RAM，磁盘空间等）。每

个任务也具有优先级，如果优先级较高的任务需要更多的资源，则可以终止（抢占）同一台机器上较低优先级的任务以释放资源。优先级还决定了计算资源的定价：团队必须为他们使用的资源付费，而优先级更高的流程花费更多【59】。

这种架构允许非生产（低优先级）计算资源被过度使用，因为系统知道如果必要的话它可以回收资源。与分离生产和非生产任务的系统相比，过度使用资源可以更好地利用机器和提高效率。但是，由于MapReduce作业以低优先级运行，因此它们随时都有被抢占的风险，因为优先级较高的进程需要其资源。批量工作有效地“拿起桌子下面的碎片”，利用高优先级进程已经采取的任何计算资源。

在谷歌，运行一个小时的MapReduce任务有大约5％被终止的风险，为更高优先级的进程腾出空间。由于硬件问题，机器重新启动或其他原因，这个速率比故障率高出一个数量级【59】。按照这种抢先率，如果一个作业有100个任务，每个任务运行10分钟，那么至少有一个任务在完成之前将被终止的风险大于50％。

这就是为什么MapReduce能够容忍频繁意外的任务终止的原因：这不是因为硬件特别不可靠，这是因为任意终止进程的自由可以在计算集群中更好地利用资源。

在开源的集群调度器中，抢占的使用较少。 YARN的CapacityScheduler支持抢占以平衡不同队列的资源分配【58】，但在编写本文时，YARN，Mesos或Kubernetes不支持通用优先级抢占【60】。在任务不经常被终止的环境中，MapReduce的设计决策没有多少意义。在下一节中，我们将看看MapReduce的一些替代方案，这些替代方案做出了不同的设计决定。

后MapReduce时代

虽然MapReduce在二十世纪二十年代后期变得非常流行并受到大量的炒作，但它只是分布式系统的许多可能的编程模型之一。根据数据量，数据结构和处理类型，其他工具可能更适合表达计算。

尽管如此，我们在讨论MapReduce的这一章花了很多时间，因为它是一个有用的学习工具，因为它是分布式文件系统的一个相当清晰和简单的抽象。也就是说，能够理解它在做什么，而不是在易于使用的意义上是简单的。恰恰相反：使用原始的MapReduce API来实现复杂的处理工作实际上是非常困难和费力的 —— 例如，你需要从头开始实现任何连接算法【37】。

针对直接使用MapReduce的困难，在MapReduce上创建了各种更高级的编程模型（Pig，Hive，Cascading，Crunch）作为抽象。如果你了解MapReduce的工作原理，那么它们相当容易学习，而且它们的高级构造使许多常见的批处理任务更容易实现。

但是，MapReduce执行模型本身也存在一些问题，这些问题并没有通过增加另一个抽象层次来解决，而且在某些类型的处理中表现得很差。一方面，MapReduce非常强大：你可以使用它来处理频繁任务终止的不可靠多租户系统上几乎任意大量的数据，并且仍然可以完成工作（虽然速度很慢）。另一方面，对于某些类型的处理来说，其他工具有时也会更快。

在本章的其余部分中，我们将介绍一些批处理方法。在第十一章我们将转向流处理，这可以看作是加速批处理的另一种方法。

内部状态表示

如前所述，每个MapReduce作业都独立于其他任何作业。作业与世界其他地方的主要联系点是分布式文件系统上的输入和输出目录。如果希望一个作业的输出成为第二个作业的输入，则需要将第二个作业的输入目录配置为与第一个作业的输出目录相同，并且外部工作流调度程序必须仅在第一份工作已经完成。

如果第一个作业的输出是要在组织内广泛发布的数据集，则此设置是合理的。在这种情况下，你需要能够通过名称来引用它，并将其用作多个不同作业（包括由其他团队开发的作业）的输入。将数据发布到分布式文件系统中的众所周知的位置允许松耦合，这样作业就不需要知道是谁在输入输出或消耗其输出（参阅“分离逻辑和布线”在本页395）。

但是，在很多情况下，你知道一个工作的输出只能用作另一个工作的输入，这个工作由同一个团队维护。在这种情况下，分布式文件系统上的文件只是简单的中间状态：一种将数据从一个作业传递到下一个作业的方式。在用于构建由50或100个MapReduce作业【29】组成的推荐系统的复杂工作流程中，存在很多这样的中间状态。

将这个中间状态写入文件的过程称为物化。（我们在“聚合：数据立方体和物化视图”中已经在物化视图的背景下遇到了这个术语。它意味着要急于计算某个操作的结果并写出来，而不是计算需要时按要求。）

相反，本章开头的日志分析示例使用Unix管道将一个命令的输出与另一个命令的输出连接起来。管道并没有完全实现中间状态，而是只使用一个小的内存缓冲区，将输出逐渐流向输入。

MapReduce的完全实现中间状态的方法与Unix管道相比存在不足：

- MapReduce作业只有在前面的作业（生成其输入）中的所有任务都完成时才能启动，而由Unix管道连接的进程同时启动，输出一旦生成就会被使用。不同机器上的偏差或不同的负荷意味着一份工作往往会有一些比其他人更快完成的离散任务。必须等到所有前面的工作完成才能减慢整个工作流程的执行。
- Mapper通常是多余的：它们只读取刚刚由reducer写入的相同文件，并为下一个分区和排序阶段做好准备。在许多情况下，Mapper代码可能是以前的reducer的一部分：如果reducer输出被分区和排序的方式与mapper输出相同，那么reducers可以直接链接在一起，而不与mapper阶段交错。
- 将中间状态存储在分布式文件系统中意味着这些文件被复制到多个节点，这对于这样的临时数据通常是过度的。

数据流引擎

了解决MapReduce的这些问题，开发了几种用于分布式批量计算的新的执行引擎，其中最著名的是Spark【61,62】，Tez【63,64】和Flink【65,66】。他们设计的方式有很多不同之处，但他们有一个共同点：他们把整个工作流作为一项工作来处理，而不是把它分解成独立的子作业。

由于它们通过几个处理阶段明确地建模数据流，所以这些系统被称为数据流引擎。像MapReduce一样，它们通过反复调用用户定义的函数来在单个线程上一次处理一条记录。他们通过对输入进行分区来并行工作，并将一个功能的输出复制到网络上，成为另一个功能的输入。

与MapReduce不同，这些功能不需要交替映射和Reduce的严格角色，而是可以以更灵活的方式进行组合。我们称之为这些函数操作符，数据流引擎提供了几个不同的选项来连接一个操作符的输出到另一个的输入：

- 一个选项是通过键对记录进行重新分区和排序，就像在MapReduce的混洗阶段一样（参阅“分布式执行MapReduce”）。此功能可以像在MapReduce中一样启用排序合并连接和分组。
- 另一种可能是采取几个输入，并以相同的方式进行分区，但跳过排序。这节省了分区散列连接的工作，其中记录的分区是重要的，但顺序是不相关的，因为构建散列表随机化了顺序。
- 对于广播散列连接，可以将一个运算符的相同输出发送到连接运算符的所有分区。

这种处理引擎的风格基于像Dryad【67】和Nephele【68】这样的研究系统，与MapReduce模型相比，它提供了几个优点：

- 排序等昂贵的工作只需要在实际需要的地方执行，而不是在每个Map和Reduce阶段之间默认发生。
- 没有不必要的Map任务，因为Mapper所做的工作通常可以合并到前面的reduce操作器中（因为Mapper不会更改数据集的分区）。
- 由于工作流程中的所有连接和数据依赖性都是明确声明的，因此调度程序会概述哪些数据是必需的，因此可以进行本地优化。例如，它可以尝试将占用某些数据的任务放在与生成它的任务相同的机器上，以便可以通过共享内存缓冲区交换数据，而不必通过网络复制数据。
- 通常将操作员之间的中间状态保存在内存中或写入本地磁盘就足够了，这比写入HDFS需要更少的I/O（必须将其复制到多个计算机并写入到每个代理的磁盘上）。MapReduce已经将这种优化用于Mapper的输出，但是数据流引擎将该思想推广到了所有的中间状态。
- 操作员可以在输入准备就绪后立即开始执行;在下一个开始之前不需要等待整个前一阶段的完成。
- 与MapReduce（为每个任务启动一个新的JVM）相比，现有的Java虚拟机（JVM）进程可以重用来运行新操作，从而减少启动开销。

你可以使用数据流引擎来执行与MapReduce工作流相同的计算，并且由于此处所述的优化，通常执行速度会明显更快。既然操作符是map和reduce的泛化，相同的处理代码可以在任一执行引擎上运行：Pig，Hive或Cascading中实现的工作流可以通过简单的配置更改从MapReduce切换到Tez或Spark，而无需修改代码【64】。

Tez是一个相当薄的库，它依赖于YARN shuffle服务来实现节点间数据的实际复制【58】，而Spark和Flink则是包含自己的网络通信层，调度器和面向用户的API的大型框架。我们将在短期内讨论这些高级API。

容错

完全实现中间状态到分布式文件系统的一个优点是它是持久的，这使得MapReduce中的容错相当容易：如果一个任务失败，它可以在另一台机器上重新启动，并从文件系统重新读取相同的输入。

Spark，Flink和Tez避免将中间状态写入HDFS，因此他们采取了不同的方法来容忍错误：如果一台机器发生故障，并且该机器上的中间状态丢失，则会从其他仍然可用的数据重新计算在可能的情况下是在先的中间阶段，或者是通常在HDFS上的原始输入数据）。

为了实现这个重新计算，框架必须跟踪一个给定的数据是如何计算的 —— 使用哪个输入分区，以及哪个操作符被应用到它。 Spark使用弹性分布式数据集（RDD）抽象来追踪数据的祖先【61】，而Flink检查点操作符状态，允许其恢复运行在执行过程中遇到错误的操作符【66】。

在重新计算数据时，重要的是要知道计算是否是确定性的：也就是说，给定相同的输入数据，操作员是否始终生成相同的输出？如果一些丢失的数据已经发送给下游运营商，这个问题就很重要。如果运营商重新启动，重新计算的数据与原有的丢失数据不一致，下游运营商很难解决新旧数据之间的矛盾。对于不确定性运营商来说，解决方案通常是杀死下游运营商，然后再运行新数据。

为了避免这种级联故障，最好让操作员具有确定性。但是请注意，非确定性行为很容易发生意外蔓延：例如，许多编程语言在迭代哈希表的元素时不能保证任何特定顺序，许多概率和统计算法明确依赖于使用随机数，以及任何用途系统时钟或外部数据源是不确定的。为了可靠地从故障中恢复，例如通过使用固定种子产生伪随机数，需要消除这种不确定性的原因。

通过重新计算数据从故障中恢复并不总是正确的答案：如果中间数据比源数据小得多，或者如果计算量非常大，那么将中间数据转化为文件可能比将其重新计算更便宜。

关于物化的讨论

回到Unix的类比，我们看到MapReduce就像是将每个命令的输出写入临时文件，而数据流引擎看起来更像是Unix管道。尤其是Flink是围绕流水线执行的思想而建立的：也就是说，将运算符的输出递增地传递给其他操作符，并且在开始处理之前不等待输入完成。

排序操作不可避免地需要消耗其整个输入，然后才能生成任何输出，因为最后一个输入记录可能是具有最低键的输入记录，因此需要作为第一个输出记录。任何需要分类的操作员都需要至少暂时地累积状态。但是工作流程的许多其他部分可以以流水线方式执行。

当作业完成时，它的输出需要持续到某个地方，以便用户可以找到并使用它—— 很可能它会再次写入分布式文件系统。因此，在使用数据流引擎时，HDFS上的物化数据集通常仍是作业的输入和最终输出。和MapReduce一样，输入是不可变的，输出被完全替换。对MapReduce的改进是，你可以节省自己将所有中间状态写入文件系统。

图与迭代处理

在“图数据模型”中，我们讨论了使用图形来建模数据，并使用图形查询语言来遍历图形中的边和顶点。第2章的讨论集中在OLTP风格的使用上：快速执行查询来查找少量符合特定条件的顶点。

在批处理环境中查看图表也很有趣，其目标是在整个图表上执行某种离线处理或分析。这种需求经常出现在机器学习应用程序（如推荐引擎）或排序系统中。例如，最着名的图形分析算法之一是PageRank 【69】，它试图根据其他网页链接的网页来估计网页的流行度。它被用作确定网络搜索引擎呈现结果的顺序的公式的一部分。

> 像Spark，Flink和Tez这样的数据流引擎（参见“中间状态的物化”）通常将操作符作为有向无环图（**DAG**）排列在作业中。这与图形处理不一样：在数据流引擎中，从一个操作符到另一个操作符的数据流被构造成一个图，而数据本身通常由关系式元组构成。在图形处理中，数据本身具有图形的形式。另一个不幸的命名混乱！

许多图算法是通过一次遍历一个边来表示的，将一个顶点与相邻的顶点连接起来以便传播一些信息，并且重复直到满足一些条件为止——例如，直到没有更多的边要跟随，或者直到一些度量收敛。我们在图2-6中看到一个例子，它通过重复地跟踪指示哪个位置在哪个其他位置（这种算法被称为传递闭包）的边缘，列出了包含在数据库中的北美所有位置。

可以在分布式文件系统（包含顶点和边的列表的文件）中存储图形，但是这种“重复直到完成”的想法不能用普通的MapReduce来表示，因为它只执行一次数据传递。这种算法因此经常以迭代方式实现：

1. 外部调度程序运行批处理来计算算法的一个步骤。
2. 当批处理过程完成时，调度器检查它是否完成（基于完成条件 - 例如，没有更多的边要跟随，或者与上次迭代相比的变化低于某个阈值）。
3. 如果尚未完成，则调度程序返回到步骤1并运行另一轮批处理。

这种方法是有效的，但是用MapReduce实现它往往是非常低效的，因为MapReduce没有考虑算法的迭代性质：它总是读取整个输入数据集并产生一个全新的输出数据集，即使只有一小部分该图与上次迭代相比已经改变。 Pregel处理模型

作为批处理图形的优化，计算的批量同步并行（BSP）模型【70】已经流行起来。其中，它由Apache Giraph 【37】，Spark的GraphX API和Flink的Gelly API 【71】实现。它也被称为Pregel模型，正如Google的Pregel论文推广这种处理图的方法【72】。

回想一下在MapReduce中，Mapper在概念上"发送消息"给reducer的特定调用，因为框架将所有的mapper输出集中在一起。 Pregel背后有一个类似的想法：一个顶点可以"发送消息"到另一个顶点，通常这些消息沿着图的边被发送。

在每次迭代中，为每个顶点调用一个函数，将所有发送给它的消息传递给它 —— 就像调用reducer一样。与MapReduce的不同之处在于，在Pregel模型中，顶点从一次迭代到下一次迭代记忆它的状态，所以这个函数只需要处理新的传入消息。如果在图的某个部分没有发送消息，则不需要做任何工作。

这与演员模型有些相似（参阅"分布式的Actor框架"），除非顶点状态和顶点之间的消息具有容错性和耐久性，并且通信以固定的方式进行，否则将每个顶点视为主角轮次：在每一次迭代中，框架传递在前一次迭代中发送的所有消息。演员通常没有这样的时间保证。

容错

顶点只能通过消息传递进行通信（而不是直接相互查询）的事实有助于提高Pregel作业的性能，因为消息可以成批处理，而且等待通信的次数也减少了。唯一的等待是在迭代之间：由于Pregel模型保证所有在一次迭代中发送的消息都在下一次迭代中传递，所以先前的迭代必须完全完成，并且所有的消息必须在网络上复制，然后下一个开始。 即使底层网络可能丢失，重复或任意延迟消息（参阅"不可靠的网络"），Pregel实施可保证在接下来的迭代中消息在其目标顶点处理一次。像MapReduce一样，该框架透明地从故障中恢复，以简化Pregel顶层算法的编程模型。

这种容错是通过在迭代结束时定期检查所有顶点的状态来实现的，即将其全部状态写入持久存储。如果某个节点发生故障并且其内存中状态丢失，则最简单的解决方法是将整个图计算回滚到上一个检查点，然后重新启动计算。如果算法是确定性的并且记录了消息，那么也可以选择性地只恢复丢失的分区（就像我们之前讨论过的数据流引擎）【72】。

并行执行

顶点不需要知道它正在执行哪个物理机器;当它发送消息到其他顶点时，它只是将它们发送到一个顶点ID。分配图的框架，即确定哪个顶点运行在哪个机器上，以及如何通过网络路由消息，以便它们结束在正确的位置。

由于编程模型一次仅处理一个顶点（有时称为"像顶点一样思考"），所以框架可以以任意方式划分图形。理想情况下，如果它们需要进行大量的通信，那么它将被分割，以使顶点在同一台机器上共置。然而，寻找这样一个优化的分割在实践中是困难的，图形经常被任意分配的顶点ID分割，而不会尝试将相关的顶点分组在一起。

因此，图算法通常会有很多跨机器通信，而中间状态（节点之间发送的消息）往往比原始图大。通过网络发送消息的开销会显着减慢分布式图算法的速度。

出于这个原因，如果你的图可以放在一台计算机的内存中，那么单机（甚至可能是单线程）算法很可能会超越分布式批处理【73,74】。即使图形大于内存，也可以放在单个计算机的磁盘上，使用GraphChi等框架进行单机处理是一个可行的选择【75】。如果图形太大而不适合单个机器，像Pregel这样的分布式方法是不可避免的。有效的并行化图算法是一个正在进行的领域。

高级API和语言

自MapReduce第一次流行以来，分布式批处理的执行引擎已经成熟。到目前为止，基础设施已经足够强大，能够存储和处理超过10,000台机器群集上的数PB的数据。由于在这种规模下物理操作批处理过程的问题已经或多或少得到了解决，所以已经转向其他领域：改进编程模型，提高处理效率，扩大这些技术可以解决的问题集。

如前所述，Hive，Pig，Cascading和Crunch等高级语言和API由于手工编写MapReduce作业而变得非常流行。随着Tez的出现，这些高级语言还有额外的好处，可以移动到新的数据流执行引擎，而无需重写作业代码。 Spark和Flink也包括他们自己的高级数据流API，经常从FlumeJava中获得灵感【34】。

这些数据流API通常使用关系式构建块来表达一个计算：连接数据集以获取某个字段的值;按键分组元组;过滤一些条件;并通过计数，求和或其他函数来聚合元组。在内部，这些操作是使用本章前面讨论过的各种连接和分组算法来实现的。

除了需要较少代码的明显优势之外，这些高级接口还允许交互式使用，在这种交互式使用中，你可以将分析代码逐步编写到shell中并经常运行，以观察它正在做什么。这种发展风格在探索数据集和试验处理方法时非常有用。这也让人联想到Unix哲学，我们在第394页的“Unix哲学”中讨论过这个问题。

而且，这些高级接口不仅使人类使用系统的效率更高，而且提高了机器级别的工作执行效率。

向声明式查询语言的转变

与拼写执行连接的代码相比，指定连接为关系运算符的优点是，框架可以分析连接输入的属性，并自动决定哪个上述连接算法最适合手头的任务。 Hive，Spark和Flink都有基于代价的查询优化器，可以做到这一点，甚至可以改变连接顺序，使中间状态的数量最小化【66,77,78,79】。

连接算法的选择可以对批处理作业的性能产生很大的影响，不必理解和记住本章中讨论的各种连接算法。如果以声明的方式指定连接，则这是可能的：应用程序简单地说明哪些连接是必需的，查询优化器决定如何最好地执行连接。我们以前在第42页的“数据的查询语言”中遇到了这个想法。

但是，在其他方面，MapReduce及其数据流后继与SQL的完全声明性查询模型有很大不同。MapReduce是围绕函数回调的思想构建的：对于每个记录或者一组记录，调用一个用户定义的函数（mapper或reducer），并且该函数可以自由地调用任意代码来决定输出什么。这种方法的优点是可以绘制在现有库的大型生态系统上进行分析，自然语言分析，图像分析以及运行数字或统计算法等。

轻松运行任意代码的自由是从MPP数据库（参见“比较Hadoop和分布式数据库”一节）中分离出来的MapReduce传统批处理系统。虽然数据库具有编写用户定义函数的功能，但是它们通常使用起来很麻烦，而且与大多数编程语言中广泛使用的程序包管理器和依赖管理系统（例如Maven for Java，npm for Java-Script，和Ruby的Ruby的Ruby）。

但是，数据流引擎已经发现，除了连接之外，在合并更多的声明性特征方面也是有优势的。例如，如果一个回调函数只包含一个简单的过滤条件，或者只是从一条记录中选择了一些字段，那么在调用每条记录的函数时会有相当大的CPU开销。如果以声明方式表示这样简单的过滤和映射操作，那么查询优化器可以利用面向列的存储布局（参阅“列存储”），并从磁盘只读取所需的列。 Hive，Spark DataFrames和Impala也使用向量化执行（参阅“内存带宽和向量处理”）：在对CPU缓存很友好的内部循环中迭代数据，并避免函数调用。

Spark生成JVM字节码【79】，Impala使用LLVM为这些内部循环生成本机代码【41】。

通过将声明性方面与高级API结合起来，并使查询优化器可以在执行期间利用这些优化方法，批处理框架看起来更像MPP数据库（并且可以实现可比较的性能）。同时，通过具有运行任意代码和以任意格式读取数据的可扩展性，它们保持了灵活性的优势。

专业化的不同领域

尽管能够运行任意代码的可扩展性是有用的，但是在标准处理模式不断重复发生的情况下也有许多常见的情况，所以值得重用通用构建块的实现。传统上，MPP数据库满足了商业智能分析师和业务报告的需求，但这只是许多使用批处理的领域之一。

另一个越来越重要的领域是统计和数值算法，它们是机器学习应用（如分类和推荐系统）所需要的。可重复使用的实现正在出现：例如，Mahout在MapReduce，Spark和Flink之上实现了用于机器学习的各种算法，而MADlib在关系型MPP数据库（Apache HAWQ）中实现了类似的功能【54】。

空间算法也是有用的，例如最近邻搜索（kNN）【80】，它在一些多维空间中搜索与给定物品接近的物品 - 这是一种类似的搜索。近似搜索对于基因组分析算法也很重要，它们需要找到相似但不相同的字符串【81】。

批处理引擎正被用于分布式执行来自日益广泛的领域的算法。随着批处理系统获得内置功能和高级声明性操作符，并且随着MPP数据库变得更加可编程和灵活，两者开始看起来更相似：最终，它们都只是存储和处理数据的系统。

本章小结

在本章中，我们探讨了批处理的主题。我们首先查看了诸如awk，grep和sort之类的Unix工具，然后我们看到了这些工具的设计理念是如何运用到MapReduce和更新的数据流引擎中的。这些设计原则中的一些是输入是不可变的，输出是为了成为另一个（还未知的）程序的输入，而复杂的问题是通过编写“做一件好事”的小工具来解决的。

在Unix世界中，允许一个程序与另一个程序组合的统一界面是文件和管道;在MapReduce中，该接口是一个分布式文件系统。我们看到数据流引擎添加了自己的管道式数据传输机制，以避免将中间状态转化为分布式文件系统，但作业的初始输入和最终输出通常仍然是HDFS。

分布式批处理框架需要解决的两个主要问题是：

分区

在MapReduce中，Mapper根据输入文件块进行分区。Mapper的输出被重新分区，分类并合并到可配置数量的reducer分区中。这个过程的目的是把所有的相关数据 —— 例如，所有的记录都放在同一个地方。

后MapReduce数据流引擎尽量避免排序，除非它是必需的，但它们采取了大致类似的分区方法。

容错

MapReduce经常写入磁盘，这使得从单个失败的任务中轻松地恢复，而无需重新启动整个作业，但在无故障的情况下减慢了执行速度。数据流引擎对中间状态执行的实现较少，并且保留在内存中，这意味着如果节点发生故障，则需要推荐更多的数据。确定性运算符减少了需要重新计算的数据量。

我们讨论了几种MapReduce的连接算法，其中大多数也是在MPP数据库和数据流引擎中使用的。他们还提供了分区算法如何工作的一个很好的例子：

排序合并连接

每个正在连接的输入都通过一个提取连接键的Mapper。通过分区，排序和合并，具有相同键的所有记录最终都会进入reducer的相同调用。这个函数可以输出连接的记录。

广播散列连接

两个连接输入之一是小的，所以它没有分区，它可以被完全加载到一个哈希表。因此，你可以为大连接输入的每个分区启动一个Mapper，将小输入的散列表加载到每个Mapper中，然后一次扫描大输入一条记录，查询每条记录的散列表。

分区散列连接

如果两个连接输入以相同的方式分区（使用相同的键，相同的散列函数和相同数量的分区），则可以独立地为每个分区使用散列表方法。

分布式批处理引擎有一个有意限制的编程模型：回调函数（比如Mapper和Reducer）被认为是无状态的，除了指定的输出外，没有外部可见的副作用。这个限制允许框架隐藏抽象背后的一些硬分布式系统问题：面对崩溃和网络问题，任务可以安全地重试，任何失败任务的输出都被丢弃。如果某个分区的多个任务成功，则只有其中一个实际上使其输出可见。

得益于这个框架，你在批处理作业中的代码无需担心实现容错机制：框架可以保证作业的最终输出与没有发生错误的情况相同，也许不得不重新尝试各种任务。这些可靠的语义要比在线服务处理用户请求时经常使用的要多得多，而且在处理请求的副作用时写入数据库。

批量处理工作的显著特点是它读取一些输入数据并产生一些输出数据，而不修改输入—— 换句话说，输出是从输入衍生出的。重要的是，输入数据是有界的：它有一个已知的，固定的大小（例如，它包含一些时间点的日志文件或数据库内容的快照）。因为它是有界的，一个工作知道什么时候它完成了整个输入的读取，所以一个工作最终完成。

在下一章中，我们将转向流处理，其中的输入是未知的 —— 也就是说，你还有一份工作，但是它的输入是永无止境的数据流。在这种情况下，工作永远不会完成，因为在任何时候都可能有更多的工作进来。我们将看到流和批处理在某些方面是相似的。但是关于无尽数据流的假设，也对我们构建系统的方式产生了很大的改变。

参考文献

1. Jeffrey Dean and Sanjay Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters,” at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.

2. Joel Spolsky: “The Perils of JavaSchools,” *joelonsoftware.com*, December 25, 2005.

3. Shivnath Babu and Herodotos Herodotou: “Massively Parallel Databases and MapReduce Systems,” *Foundations and Trends in Databases*, volume 5, number 1, pages 1–104, November 2013. doi:10.1561/1900000036

4. David J. DeWitt and Michael Stonebraker: “MapReduce: A Major Step Backwards,” originally published at *databasecolumn.vertica.com*, January 17, 2008.

5. Henry Robinson: “The Elephant Was a Trojan Horse: On the Death of Map-Reduce at Google,” *the-paper-trail.org*, June 25, 2014.

6. “The Hollerith Machine,” United States Census Bureau, *census.gov*.

7. “IBM 82, 83, and 84 Sorters Reference Manual,” Edition A24-1034-1, International Business Machines Corporation, July 1962.

8. Adam Drake: “Command-Line Tools Can Be 235x Faster than Your Hadoop Cluster,” *aadrake.com*, January 25, 2014.

9. “GNU Coreutils 8.23 Documentation,” Free Software Foundation, Inc., 2014.
10. Martin Kleppmann: “Kafka, Samza, and the Unix Philosophy of Distributed Data,” *martin.kleppmann.com*, August 5, 2015.
11. Doug McIlroy: Internal Bell Labs memo, October 1964. Cited in: Dennis M. Richie: “Advice from Doug McIlroy,” *cm.bell-labs.com*.
12. M. D. McIlroy, E. N. Pinson, and B. A. Tague: “UNIX Time-Sharing System: Foreword,” *The Bell System Technical Journal*, volume 57, number 6, pages 1899–1904, July 1978.
13. Eric S. Raymond: *The Art of UNIX Programming*. Addison-Wesley, 2003. ISBN: 978-0-13-142901-7
14. Ronald Duncan: “Text File Formats – ASCII Delimited Text – Not CSV or TAB Delimited Text,” *ronaldduncan.wordpress.com*, October 31, 2009.
15. Alan Kay: “Is 'Software Engineering' an Oxymoron?,” *tinlizzie.org*.
16. Martin Fowler: “InversionOfControl,” *martinfowler.com*, June 26, 2005.
17. Daniel J. Bernstein: “Two File Descriptors for Sockets,” *cr.yp.to*.
18. Rob Pike and Dennis M. Ritchie: “The Styx Architecture for Distributed Systems,” *Bell Labs Technical Journal*, volume 4, number 2, pages 146–152, April 1999.
19. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: “The Google File System,” at *19th ACM Symposium on Operating Systems Principles* (SOSP), October 2003. doi:10.1145/945445.945450
20. Michael Ovsiannikov, Silvius Rus, Damian Reeves, et al.: “The Quantcast File System,” *Proceedings of the VLDB Endowment*, volume 6, number 11, pages 1092–1101, August 2013. doi:10.14778/2536222.2536234
21. “OpenStack Swift 2.6.1 Developer Documentation,” OpenStack Foundation, *docs.openstack.org*, March 2016.
22. Zhe Zhang, Andrew Wang, Kai Zheng, et al.: “Introduction to HDFS Erasure Coding in Apache Hadoop,” *blog.cloudera.com*, September 23, 2015.
23. Peter Cnudde: “Hadoop Turns 10,” *yahoohadoop.tumblr.com*, February 5, 2016.
24. Eric Baldeschwieler: “Thinking About the HDFS vs. Other Storage Technologies,” *hortonworks.com*, July 25, 2012.
25. Brendan Gregg: “Manta: Unix Meets Map Reduce,” *dtrace.org*, June 25, 2013.

26. Tom White: *Hadoop: The Definitive Guide*, 4th edition. O'Reilly Media, 2015. ISBN: 978-1-491-90163-2

27. Jim N. Gray: “Distributed Computing Economics,” Microsoft Research Tech Report MSR-TR-2003-24, March 2003.

28. Márton Trencséni: “Luigi vs Airflow vs Pinball,” *bytepawn.com*, February 6, 2016.

29. Roshan Sumbaly, Jay Kreps, and Sam Shah: “The 'Big Data' Ecosystem at LinkedIn,” at *ACM International Conference on Management of Data* (SIGMOD), July 2013. doi:10.1145/2463676.2463707

30. Alan F. Gates, Olga Natkovich, Shubham Chopra, et al.: “Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience,” at *35th International Conference on Very Large Data Bases* (VLDB), August 2009.

31. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, et al.: “Hive – A Petabyte Scale Data Warehouse Using Hadoop,” at *26th IEEE International Conference on Data Engineering* (ICDE), March 2010. doi:10.1109/ICDE.2010.5447738

32. “Cascading 3.0 User Guide,” Concurrent, Inc., *docs.cascading.org*, January 2016.

33. “Apache Crunch User Guide,” Apache Software Foundation, *crunch.apache.org*.

34. Craig Chambers, Ashish Raniwala, Frances Perry, et al.: “FlumeJava: Easy, Efficient Data-Parallel Pipelines,” at *31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 2010. doi:10.1145/1806596.1806638

35. Jay Kreps: “Why Local State is a Fundamental Primitive in Stream Processing,” *oreilly.com*, July 31, 2014.

36. Martin Kleppmann: “Rethinking Caching in Web Apps,” *martin.kleppmann.com*, October 1, 2012.

37. Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira: *Hadoop Application Architectures*. O'Reilly Media, 2015. ISBN: 978-1-491-90004-8

38. Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: “Challenges to Adopting Stronger Consistency at Scale,” at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

39. Sriranjan Manjunath: “Skewed Join,” *wiki.apache.org*, 2009.

40. David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri: “Practical Skew Handling in Parallel Joins,” at *18th International Conference on Very Large Data Bases* (VLDB), August 1992.

41. Marcel Kornacker, Alexander Behm, Victor Bittorf, et al.: "Impala: A Modern, Open-Source SQL Engine for Hadoop," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

42. Matthieu Monsch: "Open-Sourcing PalDB, a Lightweight Companion for Storing Side Data," *engineering.linkedin.com*, October 26, 2015.

43. Daniel Peng and Frank Dabek: "Large-Scale Incremental Processing Using Distributed Transactions and Notifications," at *9th USENIX conference on Operating Systems Design and Implementation* (OSDI), October 2010.

44. ""Cloudera Search User Guide," Cloudera, Inc., September 2015.

45. Lili Wu, Sam Shah, Sean Choi, et al.: "The Browsemaps: Collaborative Filtering at LinkedIn," at *6th Workshop on Recommender Systems and the Social Web* (RSWeb), October 2014.

46. Roshan Sumbaly, Jay Kreps, Lei Gao, et al.: "Serving Large-Scale Batch Computed Data with Project Voldemort," at *10th USENIX Conference on File and Storage Technologies* (FAST), February 2012.

47. Varun Sharma: "Open-Sourcing Terrapin: A Serving System for Batch Generated Data," *engineering.pinterest.com*, September 14, 2015.

48. Nathan Marz: "ElephantDB," *slideshare.net*, May 30, 2011.

49. Jean-Daniel (JD) Cryans: "How-to: Use HBase Bulk Loading, and Why," *blog.cloudera.com*, September 27, 2013.

50. Nathan Marz: "How to Beat the CAP Theorem," *nathanmarz.com*, October 13, 2011.

51. Molly Bartlett Dishman and Martin Fowler: "Agile Architecture," at *O'Reilly Software Architecture Conference*, March 2015.

52. David J. DeWitt and Jim N. Gray: "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992. doi:10.1145/129888.129894

53. Jay Kreps: "But the multi-tenancy thing is actually really really hard," tweetstorm, *twitter.com*, October 31, 2014.

54. Jeffrey Cohen, Brian Dolan, Mark Dunlap, et al.: "MAD Skills: New Analysis Practices for Big Data," *Proceedings of the VLDB Endowment*, volume 2, number 2, pages 1481–1492, August 2009. doi:10.14778/1687553.1687576

55. Ignacio Terrizzano, Peter Schwarz, Mary Roth, and John E. Colino: "Data Wrangling: The Challenging Journey from the Wild to the Lake," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

56. Paige Roberts: "To Schema on Read or to Schema on Write, That Is the Hadoop Data Lake Question," *adaptivesystemsinc.com*, July 2, 2015.

57. Bobby Johnson and Joseph Adler: "The Sushi Principle: Raw Data Is Better," at *Strata+Hadoop World*, February 2015.

58. Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, et al.: "Apache Hadoop YARN: Yet Another Resource Negotiator," at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523633

59. Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, et al.: "Large-Scale Cluster Management at Google with Borg," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741964

60. Malte Schwarzkopf: "The Evolution of Cluster Scheduler Architectures," *firmament.io*, March 9, 2016.

61. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al.: "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.

62. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia: *Learning Spark*. O'Reilly Media, 2015. ISBN: 978-1-449-35904-1

63. Bikas Saha and Hitesh Shah: "Apache Tez: Accelerating Hadoop Query Processing," at *Hadoop Summit*, June 2014.

64. Bikas Saha, Hitesh Shah, Siddharth Seth, et al.: "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2742790

65. Kostas Tzoumas: "Apache Flink: API, Runtime, and Project Roadmap," *slideshare.net*, January 14, 2015.

66. Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al.: "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, volume 23, number 6, pages 939–964, May 2014. doi:10.1007/s00778-014-0357-y

67. Michael Isard, Mihai Budiu, Yuan Yu, et al.: "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," at *European Conference on Computer Systems* (EuroSys), March 2007. doi:10.1145/1272996.1273005

68. Daniel Warneke and Odej Kao: "Nephele: Efficient Parallel Data Processing in the Cloud," at *2nd Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS), November 2009. doi:10.1145/1646468.1646476

69. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd: "The PageRank

70. Leslie G. Valiant: "A Bridging Model for Parallel Computation," *Communications of the ACM*, volume 33, number 8, pages 103–111, August 1990. doi:10.1145/79173.79181

71. Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl: "Spinning Fast Iterative Data Flows," *Proceedings of the VLDB Endowment*, volume 5, number 11, pages 1268-1279, July 2012. doi:10.14778/2350229.2350245

72. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, et al.: "Pregel: A System for Large-Scale Graph Processing," at *ACM International Conference on Management of Data* (SIGMOD), June 2010. doi:10.1145/1807167.1807184

73. Frank McSherry, Michael Isard, and Derek G. Murray: "Scalability! But at What COST?," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

74. Ionel Gog, Malte Schwarzkopf, Natacha Crooks, et al.: "Musketeer: All for One, One for All in Data Processing Systems," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741968

75. Aapo Kyrola, Guy Blelloch, and Carlos Guestrin: "GraphChi: Large-Scale Graph Computation on Just a PC," at *10th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2012.

76. Andrew Lenharth, Donald Nguyen, and Keshav Pingali: "Parallel Graph Analytics," *Communications of the ACM*, volume 59, number 5, pages 78–87, May

    i. doi:10.1145/2901919

77. Fabian Hüske: "Peeking into Apache Flink's Engine Room," *flink.apache.org*, March 13, 2015.

78. Mostafa Mokhtar: "Hive 0.14 Cost Based Optimizer (CBO) Technical Overview," *hortonworks.com*, March 2, 2015.

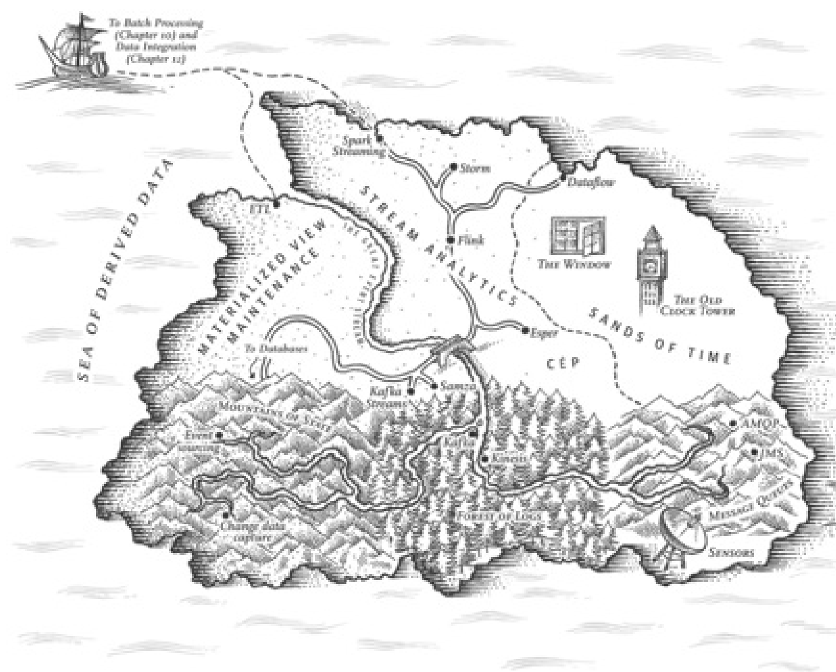
79. Michael Armbrust, Reynold S Xin, Cheng Lian, et al.: "Spark SQL: Relational Data Processing in Spark," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2742797

80. Daniel Blazevski: "Planting Quadtrees for Apache Flink," *insightdataengineering.com*, March 25, 2016.

81. Tom White: “Genome Analysis Toolkit: Now Using Apache Spark for Data Processing,” *blog.cloudera.com*, April 6, 2016.

上一章	目录	下一章
第三部分：派生数据	设计数据密集型应用	第十章：流处理

11. 流处理



> 有效的复杂系统总是从简单的系统演化而来。 反之亦然：从零设计的复杂系统没一个能有效工作的。
>
> ——约翰·加尔，Systemantics（1975）

[TOC]

在第10章中，我们讨论了批处理技术，它将一组文件读取为输入并生成一组新的输出文件。输出是派生数据的一种形式;也就是说，如果需要，可以通过再次运行批处理过程来重新创建数据集。我们看到了如何使用这个简单而强大的想法来创建搜索索引，推荐系统，分析等等。

然而，在第10章中仍然有一个大的假设：即输入是有界的，即已知和有限的大小，所以批处理知道它何时完成了它的输入。例如，MapReduce中心的排序操作必须读取其全部输入，然后才能开始生成输出：可能发生最后一个输入记录是具有最低键的输入记录，因此必须是第一个输出记录，所以提前开始输出不是一种选择。

实际上，很多数据是无限的，因为它随着时间的推移逐渐到达：你的用户昨天和今天产生了数据，明天他们将继续产生更多的数据。除非您停业，否则这个过程永远都不会结束，所以数据集从来就不会以任何有意义的方式“完成”【1】。因此，批处理程序必须将数据人为地分

成固定时间段的数据块，例如，在每天结束时处理一天的数据，或者在每小时结束时处理一小时的数据。

日常批处理过程中的问题是，输入的更改只会在一天之后的输出中反映出来，这对于许多急躁的用户来说太慢了。为了减少延迟，我们可以更频繁地运行处理 - 比如说，在每秒钟的末尾，甚至连续地处理秒数的数据，完全放弃固定的时间片，并简单地处理每一个事件。这是流处理背后的想法。

一般来说，“流”是指随着时间的推移逐渐可用的数据。这个概念出现在很多地方：Unix的stdin和stdout，编程语言（lazy lists）【2】，文件系统API（如Java的FileInputStream），TCP连接，通过互联网传送音频和视频等等。 在本章中，我们将把事件流视为一个数据管理机制：我们在上一章中看到的批量数据的无界的，递增处理的对应物。我们将首先讨论流如何被表示，存储和通过网络传输。在“数据库和流”中，我们将研究流和数据库之间的关系。最后在“流处理”中，我们将探索连续处理这些流的方法和工具，以及它们可以用来构建应用程序的方法。

传递事件流

在批处理领域，作业的输入和输出是文件（也许在分布式文件系统上）。什么是类似的流媒体？

当输入是一个文件（一个字节序列）时，第一个处理步骤通常是将其解析为一系列记录。在流处理的上下文中，记录通常被称为事件，但它本质上是一样的：一个小的，自包含的，不可变的对象，包含某个时间点发生的事情的细节。一个事件通常包含一个时间戳，指示何时根据时钟来发生（参见“单调钟与时钟”）。

例如，发生的事情可能是用户采取的行动，例如查看页面或进行购买。它也可能来源于机器，例如来自温度传感器的周期性测量或者CPU利用率度量。在“使用Unix工具进行批处理”的示例中，Web服务器日志的每一行都是一个事件。

事件可能被编码为文本字符串或JSON，或者以某种二进制形式编码，如第4章所述。这种编码允许您存储一个事件，例如将其附加到一个文件，将其插入关系表，或将其写入文档数据库。它还允许您通过网络将事件发送到另一个节点以进行处理。

在批处理领域，作业的输入和输出是文件（也许在分布式文件系统上）。什么是类似的流媒体？

当输入是一个文件（一个字节序列）时，第一个处理步骤通常是将其解析为一系列记录。在流处理的上下文中，记录通常被称为事件，但它本质上是一样的：一个小的，自包含的，不可变的对象，包含某个时间点发生的事情的细节。一个事件通常包含一个时间戳，指示何时根据时钟来发生（参见“单调钟与时钟”）。

例如，发生的事情可能是用户采取的行动，例如查看页面或进行购买。它也可能来源于机器，例如来自温度传感器的周期性测量或者CPU利用率度量。在“使用Unix工具进行批处理”的示例中，Web服务器日志的每一行都是一个事件。

事件可能被编码为文本字符串或JSON，或者以某种二进制形式编码，如第4章所述。这种编码允许您存储一个事件，例如将其附加到一个文件，将其插入关系表，或将其写入文档数据库。它还允许您通过网络将事件发送到另一个节点以进行处理。

在批处理中，文件被写入一次，然后可能被多个作业读取。类似地，在流媒体术语中，一个事件由生产者（也称为发布者或发送者）生成一次，然后由多个消费者（订阅者或接收者）进行处理【3】。在文件系统中，文件名标识一组相关记录;在流媒体系统中，相关的事件通常被组合成一个主题或流。

原则上，文件或数据库足以连接生产者和消费者：生产者将其生成的每个事件写入数据存储，并且每个使用者定期轮询数据存储以检查自上次运行以来出现的事件。这实际上是批处理在每天结束时处理一天的数据的过程。

但是，如果数据存储不是为这种用途而设计的，那么在延迟较小的情况下继续进行处理时，轮询将变得非常昂贵。您调查的次数越多，返回新事件的请求百分比越低，因此开销越高。相反，当新事件出现时，最好通知消费者。

数据库传统上不太支持这种通知机制：关系数据库通常具有触发器，它们可以对变化作出反应（例如，将一行插入到表中），但是它们的功能非常有限，有点在数据库设计中是事后的【4,5】。相反，已经开发了专门的工具来提供事件通知。

消息系统

通知消费者有关新事件的常用方法是使用消息传递系统：生产者发送包含事件的消息，然后将消息推送给消费者。我们之前在“消息传递中的数据流”中介绍了这些系统，但现在我们将详细介绍这些系统。

像生产者和消费者之间的Unix管道或TCP连接这样的直接通信渠道将是实现消息传递系统的简单方法。但是，大多数消息传递系统都在这个基本模型上扩展特别是，Unix管道和TCP将一个发送者与一个接收者完全连接，而一个消息传递系统允许多个生产者节点将消息发送到相同的主题，并允许多个消费者节点接收主题中的消息。

在这个发布/订阅模式中，不同的系统采取多种方法，对于所有目的都没有一个正确的答案。为了区分这些系统，询问以下两个问题特别有帮助：

1. 如果生产者发送消息的速度比消费者能够处理的速度快，会发生什么？一般来说，有三种选择：系统可以放置消息，缓冲队列中的消息或应用背压（也称为流量控制;即阻止生产者发送更多的消息）。例如，Unix管道和TCP使用背压：他们有一个小的固定大小的缓冲区，如果填满，发件人被阻塞，直到收件人从缓冲区中取出数据（参见“网络拥塞和排队”）。

如果消息被缓存在队列中，那么了解该队列增长会发生什么很重要。如果队列不再适合内存，或者将消息写入磁盘，系统是否会崩溃？如果是这样，磁盘访问如何影响邮件系统的性能【6】？

2. 如果节点崩溃或暂时脱机，会发生什么情况 - 是否有消息丢失？与数据库一样，持久性可能需要写入磁盘和/或复制的某种组合（参阅“复制和持久性”），这有成本。如果您有时可能会丢失消息，则可能在同一硬件上获得更高的吞吐量和更低的延迟。

消息丢失是否可以接受取决于应用程序。例如，对于定期传输的传感器读数和指标，偶尔丢失的数据点可能并不重要，因为更新后的值将在短时间后发送。但是，要注意，如果大量的消息被丢弃，那么衡量标准是不正确的【7】。如果您正在计数事件，那么更重要的是它们可靠地传送，因为每个丢失的消息都意味着不正确的计数器。

我们在第10章中探讨的批处理系统的一个很好的特性是它们提供了强大的可靠性保证：失败的任务会自动重试，失败任务的部分输出会自动丢弃。这意味着输出与没有发生故障一样，这有助于简化编程模型。在本章的后面，我们将研究如何在流式上下文中提供类似的保证。

直接从生产者传递给消费者

许多消息传递系统使用生产者和消费者之间的直接网络通信，而不通过中间节点：

- UDP组播广泛应用于金融行业，例如股票市场，其中低时延非常重要【8】。虽然UDP本身是不可靠的，但应用程序级别的协议可以恢复丢失的数据包（生产者必须记住它发送的数据包，以便它可以按需重新发送数据包）。

- 无代理的消息库，如ZeroMQ 【9】和nanomsg采取类似的方法，通过TCP或IP多播实现发布/订阅消息传递。

  StatsD 【10】和Brubeck 【7】使用不可靠的UDP消息传递来收集网络中所有机器的指标并对其进行监控。 （在StatsD协议中，如果接收到所有消息，则计数器度量标准是正确的;使用UDP将使得度量标准最好近似为【11】。另请参阅“TCP与UDP”

- 如果消费者在网络上公开服务，生产者可以直接发送HTTP或RPC请求（参阅“通过服务进行数据流：REST和RPC”）以将消息推送给使用者。这就是webhooks背后的想法【12】，一种服务的回调URL被注册到另一个服务中，并且每当事件发生时都会向该URL发出请求。

尽管这些直接消息传递系统在设计它们的情况下运行良好，但是它们通常要求应用程序代码知道消息丢失的可能性。他们可以容忍的错误是相当有限的：即使协议检测并重新传输在网络中丢失的数据包，他们通常假设生产者和消费者不断在线。

如果消费者处于脱机状态，则可能会丢失在无法访问时发送的消息。一些协议允许生产者重试失败的消息传递，但是如果生产者崩溃，这种方法可能会失效，失去了它应该重试的消息的缓冲区。

消息代理

一个广泛使用的替代方法是通过消息代理（也称为消息队列）发送消息，消息代理实质上是一种针对处理消息流而优化的数据库。它作为服务器运行，生产者和消费者作为客户连接到服务器。生产者将消息写入经纪人，消费者通过从经纪人那里读取消息来接收他们。

通过将数据集中在代理中，这些系统可以更容易地容忍来来去去的客户端（连接，断开连接和崩溃），而耐久性问题则转移到代理商。一些消息代理只将消息保存在内存中，而另一些消息代理（取决于配置）将其写入磁盘，以便在代理崩溃的情况下不会丢失。面对缓慢的消费者，他们通常会允许排除异常的排队（而不是丢弃消息或背压），尽管这种选择也可能取决于配置。

排队的结果也是消费者通常是异步的：当生产者发送消息时，通常只等待代理确认已经缓存消息，不等待消息被消费者处理。向消费者的交付将发生在未来某个未定的时间点 - 通常在几分之一秒之内，但有时在队列积压之后显着延迟。

消息代理与数据库进行比较

有些消息代理甚至可以使用XA或JTA参与两阶段提交协议（参阅“实践中的分布式事务”）。这个功能与数据库在本质上非常相似，尽管消息代理和数据库之间仍存在重要的实际差异：

- 数据库通常保留数据，直到被明确删除，而大多数消息代理在消息成功传递给消费者时自动删除消息。这样的消息代理不适合长期的数据存储。
- 由于他们很快删除了邮件，大多数邮件经纪人都认为他们的工作集合相当小，即队列很短。如果代理需要缓冲很多消息，因为消费者速度较慢（如果消息不再适合内存，则可能会将消息泄漏到磁盘），每个消息需要更长的时间处理，整体吞吐量可能会降低【6】。
- 数据库通常支持二级索引和各种搜索数据的方式，而消息代理通常支持某种方式订阅匹配某种模式的主题子集。机制是不同的，但是这两种方式都是客户选择想要了解的数据部分的根本途径。
- 查询数据库时，结果通常基于数据的时间点快照;如果另一个客户端随后向数据库写入更改查询结果的内容，则第一个客户端不会发现其先前的结果现在已过期（除非它重复查询或轮询更改）。相比之下，消息代理不支持任意查询，但是当数据发生变化时（即新消息可用时），它们会通知客户端。

这是消息代理的传统观点，它被封装在JMS 【14】和AMQP 【15】等标准中，并以RabbitMQ，ActiveMQ，HornetQ，Qpid，TIBCO企业消息服务，IBM MQ，Azure服务总线和Google Cloud Pub/Sub 【16】。

多个消费者

当多个消费者阅读同一主题中的消息时，将使用两种主要的消息传递模式，如图11-1所示：

负载均衡

每条消息都被传递给其中一个消费者，所以消费者可以共享处理消息的主题。经纪人可以任意分配消息给消费者。当处理消息的代价很高时，此模式非常有用，因此您希望能够添加消费者来并行处理消息。（在AMQP中，可以通过让多个客户端从同一个队列中消费来实现负载均衡，而在JMS中则称为共享订阅）。

扇出

每条消息都被传递给所有的消费者。扇出允许几个独立的消费者每个“收听”相同的消息广播，而不会相互影响 - 流式等同于具有读取相同输入文件的多个不同批处理作业。（此功能由JMS中的主题订阅提供，并在AMQP中交换绑定。）

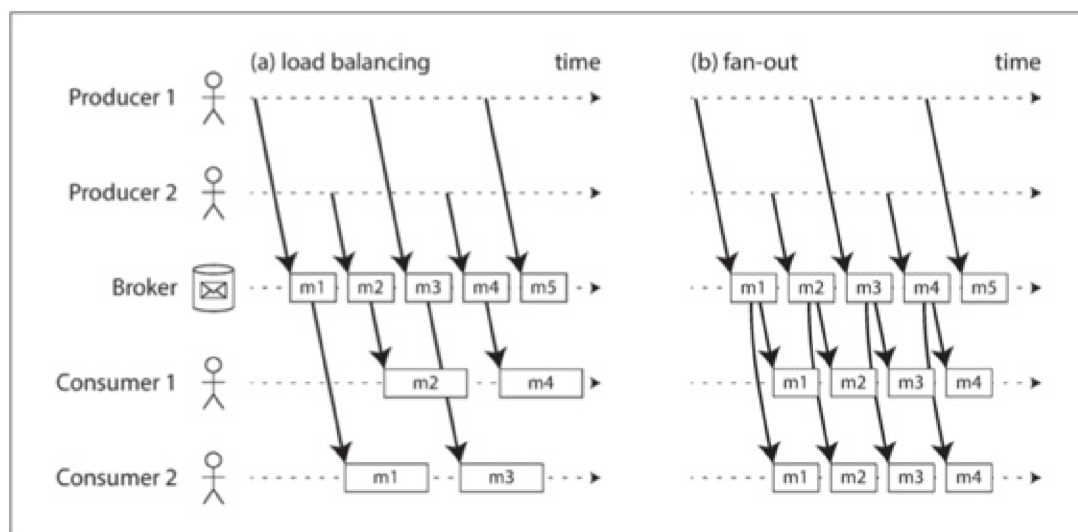


图**11-1**。（**a**）负载平衡：分担消费者之间消费话题的工作;（**b**）扇出：将消息传递给多个消费者。

两种模式可以组合：例如，两个独立的消费者组可以每个订阅一个话题，使得每个组共同收到所有消息，但是在每个组内，只有一个节点接收每个消息。

确认和重新交付

消费者可能会随时崩溃，所以可能发生的情况是经纪人向消费者提供消息，但消费者从不处理消费者，或者在消费者崩溃之前只处理消费者。为了确保消息不会丢失，消息代理使用确认：客户端必须明确告诉代理处理消息的时间，以便代理可以将其从队列中移除。

如果与客户端的连接关闭或超时而没有代理收到确认，则认为消息未处理，因此它将消息再次传递给另一个消费者。（请注意，可能发生这样的消息实际上是完全处理的，但网络中的确认丢失了，处理这种情况需要一个原子提交协议，正如在“实践中的分布式事务”中所讨论的那样）

当与负载平衡相结合时，这种重新传递行为对消息的排序有一个有趣的影响。在图11-2中，消费者通常按照生产者发送的顺序处理消息。然而，消费者2在处理消息m3时崩溃，与此同时消费者1正在处理消息m4。未确认的消息m3随后被重新发送给消费者1，结果消费者1按照

m4，m3，m5的顺序处理消息。因此，m3和m4不是以与生产者1发送的顺序相同的顺序交付的。

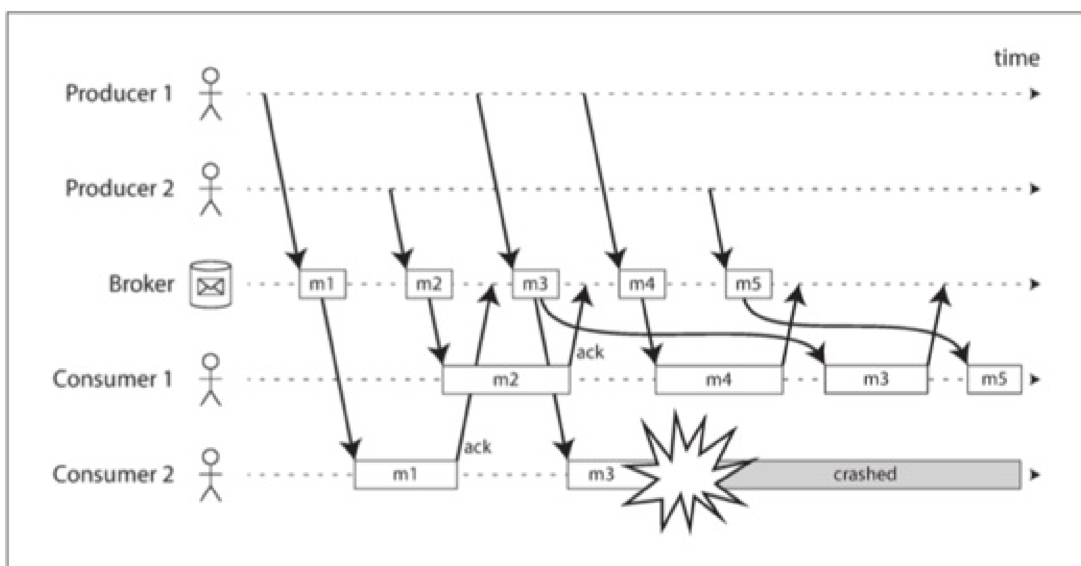


图11-2 消费者2在处理m3时崩溃，因此稍后再次向消费者1递送

即使消息代理试图保留消息的顺序（如JMS和AMQP标准所要求的），负载平衡与重新传递的组合也不可避免地导致消息被重新排序。为避免此问题，您可以为每个使用者使用单独的队列（即不使用负载均衡功能）。如果消息是完全独立的，消息重新排序并不是一个问题，但是如果消息之间存在因果依赖关系，这一点很重要，我们将在后面的章节中看到。

分区日志

通过网络发送数据包或向网络服务发送请求通常是暂时的操作，不会留下永久的痕迹。尽管可以永久记录（使用数据包捕获和日志记录），但我们通常不这么想。即使是信息经纪人，他们将信息持久地写入磁盘，在被传递给消费者之后，很快就会将其删除，因为它们是以短暂的消息传递思想为基础构建的。

数据库和文件系统采用相反的方法：写入数据库或文件的所有内容通常都要被永久记录下来，至少在某些人明确选择将其删除之前。

思维方式上的这种差异对如何创建派生数据有很大的影响。批处理过程的一个关键特征是，你可以反复运行它们，试验处理步骤，没有损坏输入的风险（因为输入是只读的）。AMQP/JMS风格的消息并非如此：如果确认导致从代理中删除消息，则接收消息具有破坏性，因此您不能再次运行同一消费者，并期望得到相同的结果。

如果您将新消费者添加到消息系统，则通常只会开始接收在注册后发送的消息;任何先前的消息已经消失，无法恢复。将它与文件和数据库进行对比，您可以随时添加新客户端，并且可以读取过去任意写入的数据（只要应用程序没有明确覆盖或删除数据）。

为什么我们不能混合使用数据库的持久存储方式和低延迟的消息通知功能呢？这是基于日志消息代理的思想。

使用日志进行消息存储

日志只是磁盘上只能追加记录的序列。我们先前在第3章中的日志结构存储引擎和预写日志的上下文中讨论了日志，在第5章中也讨论了复制的上下文中。

可以使用相同的结构来实现消息代理：生产者通过将消息附加到日志的末尾来发送消息，并且消费者通过依次读取日志来接收消息。如果消费者到达日志的末尾，则等待通知新消息已被追加。 Unix工具 `tail -f` 监视数据被附加的文件，基本上是这样工作的。

为了扩展到比单个磁盘提供更高的吞吐量，可以对日志进行分区（在第6章的意义上）。然后可以在不同的机器上托管不同的分区，使每个分区成为一个单独的日志，可以独立于其他分区读取和写入。然后可以将一个话题定义为一组分段，它们都携带相同类型的消息。这种方法如图11-3所示。

在每个分区中，代理为每个消息分配一个单调递增的序列号或偏移量（在图11-3中，框中的数字是消息偏移量）。这样的序列号是有意义的，因为分区是仅附加的，所以分区内的消息是完全有序的。没有跨不同分区的订购保证。

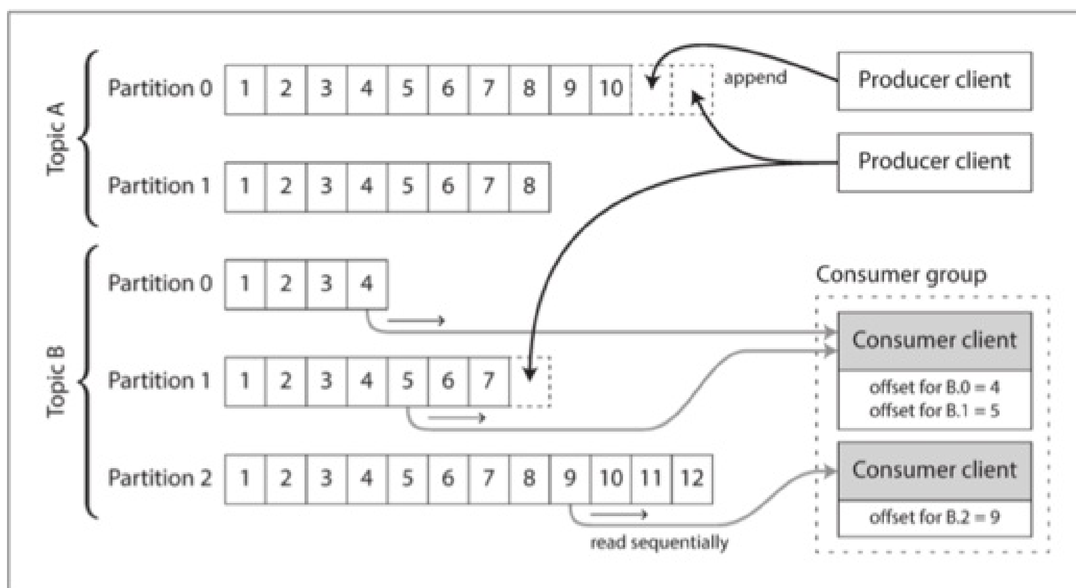


图**11-3** 生产者通过将消息附加到主题分区文件来发送消息，消费者依次读取这些文件

Apache Kafka 【17,18】，Amazon Kinesis Streams 【19】和Twitter的DistributedLog 【20,21】都是基于日志的消息代理。 Google Cloud Pub / Sub在体系结构上相似，但是暴露了JMS风格的API而不是日志抽象【16】。尽管这些消息代理将所有消息写入磁盘，但通过在多台机器上进行分区，每秒能够实现数百万条消息的吞吐量，并通过复制消息来实现容错性【22,23】。

日志与传统消息相比

基于日志的方法平凡支持扇出消息传递，因为多个消费者可以独立读取日志，而不会相互影响 - 读取消息不会将其从日志中删除。为了在一组消费者之间实现负载平衡，代理人可以将整个分区分配给消费者组中的节点，而不是将消息分配给消费者客户端。

每个客户端然后使用分配的分区中的所有消息。通常情况下，当一个用户被分配了一个日志分区时，它会以直接的单线程的方式顺序地读取分区中的消息。这种粗粒度的负载均衡方法有一些缺点：

- 共享消费主题工作的节点数最多可以是该主题中的日志分区数，因为同一个分区内的消息被传递到同一个节点[i]。
- 如果单个消息处理缓慢，则会阻止处理该分区中的后续消息（一种行头阻塞形式；请参阅“描述性能”）。

因此，在处理消息可能代价高昂，并且希望逐个消息地平行处理以及消息排序并不那么重要的情况下，消息代理的JMS/AMQP风格是可取的。另一方面，在消息吞吐量高的情况下，每条消息的处理速度都很快，消息的排序也是重要的，基于日志的方法运行得很好。

> i. 有可能创建一个负载均衡方案，在这个方案中，两个消费者通过读取全部消息来共享处理分区的工作，但是其中一个只考虑具有偶数偏移量的消息，而另一个消费者处理奇数编号的偏移量。或者，您可以将消息处理扩展到线程池，但这种方法会使消费者偏移管理变得复杂。一般来说，分区的单线程处理是可取的，并行分区可以通过使用更多的分区来增加。 ↩

消费者偏移量

消耗一个分区依次可以很容易地判断哪些消息已经被处理：所有消息的偏移量小于消费者的当前偏移量已经被处理，并且具有更大偏移量的所有消息还没有被看到。因此，经纪人不需要跟踪每条消息的确认，只需要定期记录消费者的偏移。在这种方法中减少的簿记开销以及批处理和流水线的机会有助于提高基于日志的系统的吞吐量。

实际上，这种偏移量与单领先数据库复制中常见的日志序列号非常相似，我们在“设置新从库”中讨论了这种情况。在数据库复制中，日志序列号允许跟随者断开连接后，重新连接到领导，并在不跳过任何写入的情况下恢复复制。这里使用的原则完全相同：消息代理的行为像一个领导者数据库，而消费者就像一个追随者。

如果消费者节点失败，则使用者组中的另一个节点将被分配失败的使用者分区，并开始以最后记录的偏移量使用消息。如果消费者已经处理了后续的消息，但还没有记录他们的偏移量，那么在重新启动后这些消息将被第二次处理。本章后面我们将讨论处理这个问题的方法。

磁盘空间使用情况

如果您只附加到日志，则最终将耗尽磁盘空间。为了回收磁盘空间，日志实际上被分割成段，并且不时地将旧段删除或移动到归档存储器。（我们将在后面讨论一种更为复杂的释放磁盘空间的方法。）

这就意味着，如果一个消费者的速度慢，消费者的消费速度落后于消费者的偏移量，那么消费者的偏移量就会指向一个已经被删除的消费者。实际上，日志实现了一个有限大小的缓冲区，当旧的消息满时，它也被称为循环缓冲区或环形缓冲区。但是，由于该缓冲区在磁盘上，因此可能相当大。

让我们来做一个后台计算。在撰写本文时，典型的大型硬盘驱动器容量为6TB，顺序写入吞吐量为150MB/s。如果您以最快的速度写邮件，则需要大约11个小时才能填满驱动器。因此，磁盘可以缓存11个小时的消息，之后它将开始覆盖旧的消息。即使您使用多个硬盘驱动器和机器，这个比率也是一样的。在实践中，部署很少使用磁盘的完整写入带宽，所以日志通常可以保存几天甚至几周的缓冲区。

不管你保留多长时间的消息，一个日志的吞吐量或多或少保持不变，因为无论如何每个消息都被写入磁盘【18】。这种行为与将邮件默认保存在内存中的消息传递系统形成鲜明对比，如果队列变得太大，只将其写入磁盘：当这些系统开始写入磁盘时，这些系统速度很快，并且变慢得多，所以吞吐量取决于保留的历史数量。

当消费者跟不上生产者时

在“信息系统”第441页的开头，我们讨论了如果消费者无法跟上生产者发送信息的速度的三种选择：丢弃信息，缓冲或施加背压。在这个分类法中，基于日志的方法是一种缓冲形式，具有较大但固定大小的缓冲区（受可用磁盘空间的限制）。

如果消费者远远落后于它所要求的信息比保留在磁盘上的信息要旧，那么它将不能读取这些信息，所以代理人有效地丢弃了比缓冲区容量更大的旧信息。您可以监控消费者在日志头后面的距离，如果显着落后，则会发出警报。由于缓冲区很大，因此有足够的时间让人类操作员修复缓慢的消费者，并在消息开始丢失之前让其赶上。

即使消费者落后太多，开始丢失信息，也只有消费者受到影响;它不会中断其他消费者的服务。这是一个巨大的运营优势：您可以通过实验性的方式使用生产日志进行开发，测试或调试，而不必担心会中断生产服务。当消费者关闭或崩溃时，会停止消耗资源，唯一剩下的就是消费者抵消。

这种行为也与传统的信息经纪人形成了鲜明的对比，在这种情况下，您需要小心删除消费者已经关闭的任何队列，否则他们会继续不必要地积累消息，并从仍然活跃的消费者那里带走内存。

重播旧信息

我们之前提到，使用AMQP和JMS风格的消息代理，处理和确认消息是一个破坏性的操作，因为它会导致消息在代理上被删除。另一方面，在基于日志的消息代理中，使用消息更像是从文件中读取数据：这是只读操作，不会更改日志。

处理的唯一副作用，除了消费者的任何产出之外，消费者补偿正在向前发展。但是偏移量是在消费者的控制之下的，所以如果需要的话可以很容易地被操纵：例如，你可以用昨天的偏移量开始一个消费者的副本，并将输出写到不同的位置，以便重新处理最后一天值得的消息。您可以重复这个任意次数，改变处理代码。

这方面使得基于日志的消息传递更像上一章的批处理过程，其中派生数据通过可重复的转换过程与输入数据明确分离。它允许更多的实验，更容易从错误和错误中恢复，使其成为在组织内集成数据流的好工具【24】。

流与数据库

我们已经在消息代理和数据库之间进行了一些比较。尽管传统上它们被视为单独的工具类别，但是我们看到基于日志的消息中介已经成功地从数据库中获取了想法并将其应用于消息传递。我们也可以反过来：从消息和流中获取想法，并将它们应用于数据库。

我们之前曾经说过，事件是某个时刻发生的事情的记录。发生的事情可能是用户操作（例如键入搜索查询）或传感器读取，但也可能是写入数据库。事情被写入数据库的事实是可以被捕获，存储和处理的事件。这一观察结果表明，数据库和数据流之间的连接不仅仅是磁盘上日志的物理存储 - 这是非常重要的。

事实上，复制日志（参阅“复制日志的实现”）是数据库写入事件的流，由领导者在处理事务时生成。追随者将写入流应用到他们自己的数据库副本，从而最终得到相同数据的精确副本。复制日志中的事件描述发生的数据更改。

我们还在“全序广播”中遇到了状态机复制原理，其中指出：如果每个事件代表对数据库的写入，并且每个副本按相同的顺序处理相同的事件，则副本将所有这些都以相同的最终状态结束。（处理一个事件被认为是一个确定性的操作。）这只是事件流的另一种情况！

在本节中，我们将首先看看异构数据系统中出现的一个问题，然后探讨如何通过将事件流的想法带入数据库来解决这个问题。

保持系统同步

正如我们在本书中所看到的，没有一个系统能够满足所有的数据存储，查询和处理需求。在实践中，大多数不重要的应用程序需要结合多种不同的技术来满足他们的需求：例如，使用OLTP数据库来为用户请求提供服务，缓存来加速常见请求，处理全文索引搜索查询和用于分析的数据仓库。每个数据都有其自己的数据副本，存储在自己的表示中，并根据自己的目的进行了优化。

由于相同或相关的数据出现在不同的地方，因此需要保持相互同步：如果某个项目在数据库中进行了更新，则还需要在缓存，搜索索引和数据仓库中进行更新。对于数据仓库，这种同步通常由ETL进程执行（参见“数据仓库”），通常通过获取数据库的完整副本，转换数据库并

将其批量加载到数据仓库中 —— 换句话说，一个批处理。同样，我们在“批量工作流的输出”中看到了如何使用批处理过程创建搜索索引，建议系统和其他派生数据系统。

如果周期性的完整数据库转储过于缓慢，有时使用的替代方法是双重写入，其中应用程序代码在数据更改时明确写入每个系统：例如，首先写入数据库，然后更新搜索索引，然后使缓存条目无效（或者甚至同时执行这些写入）。

但是，双重写入有一些严重的问题，其中一个是图11-4所示的竞争条件。在这个例子中，两个客户端同时想要更新一个项目X：客户端1想要将值设置为A，客户端2想要将其设置为B.两个客户端首先将新值写入数据库，然后将其写入到搜索索引。由于运行时间不正确，这些请求是交错的：数据库首先看到从客户端1的写入将值设置为A，然后从客户端2写入将值设置为B，因此数据库中的最终值为B.搜索索引首先看到来自客户端2，然后是客户端1的写入，所以搜索索引中的最终值是A.这两个系统现在永久地不一致，即使没有发生错误。

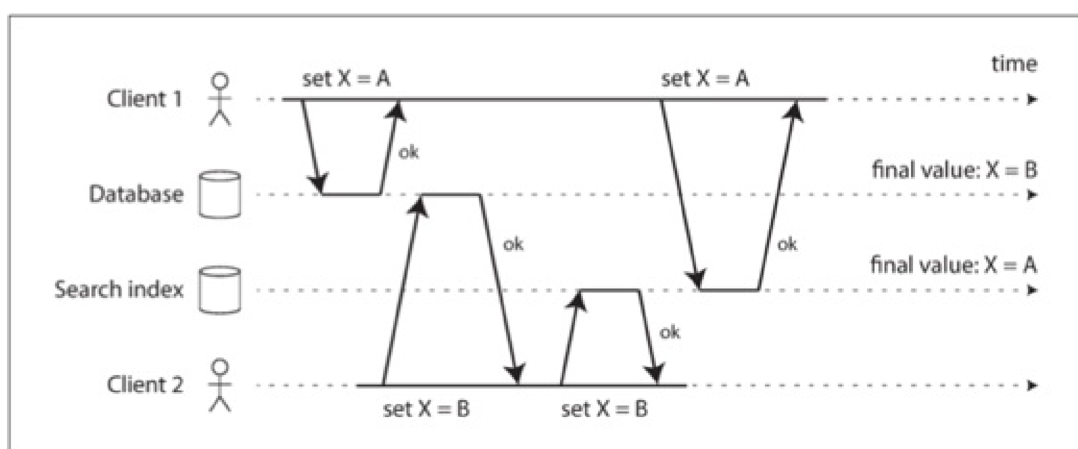


图**11-4** 在数据库中，**X**首先被设置为**A**，然后被设置为**B**，而在搜索索引处，写入以相反的顺序到达

除非有一些额外的并发检测机制，例如我们在“检测并发写入”中讨论的版本向量，否则您甚至不会注意到发生了并发写入 —— 一个值将简单地以无提示方式覆盖另一个值。

双重写入的另一个问题是其中一个写入可能会失败，而另一个成功。这是一个容错问题，而不是一个并发问题，但也会造成两个系统互相矛盾的结果。确保它们都成功或者两者都失败是原子提交问题的一个例子，这个问题的解决是昂贵的（参阅“原子提交和两阶段提交（2PC）”）。

如果只有一个复制的数据库和一个领导者，那么这个领导者决定了写入顺序，所以状态机复制方法可以在数据库的副本中工作。然而，在图11-4中，没有一个领导者：数据库可能有一个领导者，搜索索引可能有一个领导者，但是既不在另一个领导者之后，也可能发生冲突（参见“多领导者复制“）。

如果实际上只有一个领导者（例如数据库），并且如果我们可以使搜索索引成为数据库的追随者，情况会更好。但这在实践中可能吗？

改变数据捕获

大多数数据库的复制日志的问题在于，它们一直被认为是数据库的内部实现细节，而不是公共API。客户端应该通过其数据模型和查询语言来查询数据库，而不是分析复制日志并尝试从中提取数据。

数十年来，许多数据库根本没有记录方式来获取写入到他们的变更日志。由于这个原因，很难将数据库中所做的所有更改复制到不同的存储技术，如搜索索引，缓存或数据仓库。

最近，人们对变更数据捕获（CDC）越来越感兴趣，它是观察写入数据库的所有数据变化并将其提取出来，并将其复制到其他系统中的过程。 CDC特别感兴趣的是，如果改变可以立即用于流，可以立即写入。

例如，您可以捕获数据库中的更改并不断将相同的更改应用于搜索索引。如果更改的日志以相同的顺序应用，则可以预期搜索索引中的数据与数据库中的数据匹配。搜索索引和任何其他派生的数据系统只是变化流的消费者，如图11-5所示。

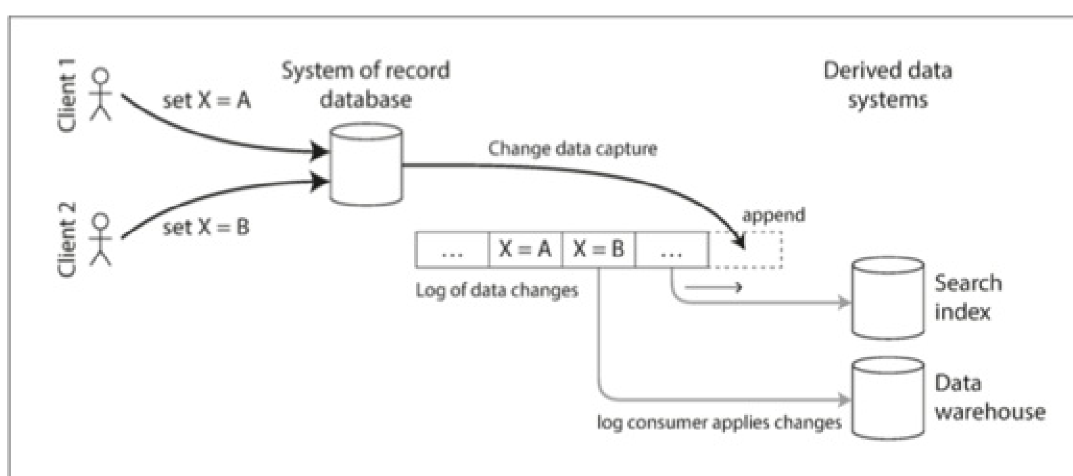


图**11-5** 将数据按顺序写入一个数据库，然后按照相同的顺序将这些更改应用到其他系统

实现变更数据捕获

我们可以调用日志消费者导出的数据系统，正如在第三部分的介绍中所讨论的：存储在搜索索引和数据仓库中的数据只是记录系统中数据的另一个视图。更改数据捕获是一种机制，可确保对记录系统所做的所有更改都反映在派生数据系统中，以便派生系统具有数据的准确副本。

从本质上说，改变数据捕获使得一个数据库成为领导者（从中捕获变化的数据库），并将其他人变成追随者。基于日志的消息代理非常适合从源数据库传输更改事件，因为它保留了消息的排序（避免了图11-2的重新排序问题）。

数据库触发器可用于通过注册触发器来实现更改数据捕获（参阅“基于触发器的复制”），这些触发器可观察数据表的所有更改，并将相应的条目添加到更改日志表中。但是，他们往往是脆弱的，并有显着的性能开销。解析复制日志可以是一个更强大的方法，但它也带来了挑战，例如处理模式更改。

LinkedIn的Databus【25】，Facebook的Wormhole【26】和Yahoo！的Sherpa【27】大规模地使用这个想法。 Bottled Water使用解码预写日志的API来实现PostgreSQL的CDC【28】，Maxwell和Debezium通过解析binlog为MySQL做类似的事情【29,30,31】，Mongoriver读取MongoDB oplog【32,33】，而GoldenGate为Oracle提供类似的功能【34,35】。

像消息代理一样，更改数据捕获通常是异步的：记录数据库系统不会等待更改应用到消费者，然后再进行更改。这种设计具有的操作优势是添加缓慢的使用者不会影响记录系统太多，但是它具有所有复制滞后问题的缺点（参见“复制延迟问题”）。

初始快照

如果具有对数据库所做的所有更改的日志，则可以通过重播日志来重新构建数据库的整个状态。但是，在许多情况下，永远保留所有更改将需要太多的磁盘空间，并且重播将花费太长时间，因此日志需要被截断。

例如，构建新的全文索引需要整个数据库的完整副本 - 仅仅应用最近更改的日志是不够的，因为它将丢失最近未更新的项目。因此，如果您没有完整的日志历史记录，则需要从一致的快照开始，如先前在第155页上的“设置新的追随者”中所述。

数据库的快照必须与更改日志中的已知位置或偏移量相对应，以便您知道在快照处理完成后，在哪一点开始应用更改。一些CDC工具集成了这个快照工具，而其他工具则将其作为手动操作。

日志压缩

如果只能保留有限的日志历史记录，则每次需要添加新的派生数据系统时都需要执行快照过程。但是，日志压缩提供了一个很好的选择。

在日志结构化的存储引擎的情况下，我们先讨论了“Hash索引”中的日志压缩（参见图3-2的示例）。原理很简单：存储引擎使用相同的密钥定期查找日志记录，丢弃任何重复内容，并且只保留每个密钥的最新更新。这个压缩和合并过程在后台运行。

在日志结构存储引擎中，具有特殊空值（逻辑删除）的更新指示删除了一个密钥，并在日志压缩过程中将其删除。但只要密钥不被覆盖或删除，它就永远留在日志中。这种压缩日志所需的磁盘空间仅取决于数据库的当前内容，而不取决于数据库中发生的写入次数。如果相同的密钥经常被覆盖，则以前的值将最终被垃圾收集，并且只保留最新的值。

在基于日志的消息代理和更改数据捕获方面，相同的想法也适用。如果CDC系统设置为每个更改都有一个主键，并且每个键的更新都替换了该键的以前的值，那么仅保留最近写入的特定键就足够了。

现在，无论何时要重建派生数据系统（如搜索索引），都可以从日志压缩主题的偏移量0开始新的使用者，然后依次扫描日志中的所有消息。日志保证包含数据库中每个键的最新值（也可能是一些较旧的值）—— 换句话说，您可以使用它来获取数据库内容的完整副本，而无需

获取CDC的另一个快照源数据库。

Apache Kafka支持此日志压缩功能。正如我们将在本章后面看到的，它允许消息代理被用于持久存储，而不仅仅是用于临时消息。

API支持更改流

越来越多的数据库开始支持变更流作为一流的接口，而不是典型的改造和逆向工程CDC的努力。例如，RethinkDB允许查询在查询更改结果【36】，Firebase 【37】和CouchDB 【38】基于同样可用于应用程序的更改提要进行数据同步时订阅通知，而Meteor使用MongoDB oplog订阅数据更改并更新用户界面【39】。

VoltDB允许事务以流的形式连续地从数据库中导出数据【40】。数据库将关系数据模型中的输出流表示为一个表，事务可以在其中插入元组，但不能被查询。然后这个流由提交事务已经写入这个特殊表的元组日志组成，它们按照提交的顺序。外部使用者可以异步使用此日志并使用它来更新派生的数据系统。

Kafka Connect 【41】致力于将广泛的数据库系统的变更数据捕获工具与Kafka集成。一旦更改事件发生在Kafka中，它就可以用来更新派生的数据系统，比如搜索索引，也可以用于本章稍后讨论的流处理系统。

事件源

我们在这里讨论的想法和事件采购之间有一些相似之处，这是一个在领域驱动设计（DDD）社区中开发的技术。我们将简要讨论事件源，因为它包含了一些有用的和相关的流式系统的想法。

与更改数据捕获类似，事件采购涉及将所有对应用程序状态的更改存储为更改事件的日志。最大的区别是事件源代码在不同的抽象层次上应用了这个想法：

- 在更改数据捕获中，应用程序以可变方式使用数据库，随意更新和删除记录。从数据库中提取较低级别的更改日志（例如，通过解析复制日志），从而确保从数据库中提取的写入顺序与实际写入的顺序相匹配，从而避免图11-4中的竞争条件。写入数据库的应用程序不需要知道CDC正在发生。
- 在事件源中，应用程序逻辑是基于写入事件日志的不可变事件而显式构建的。在这种情况下，事件存储是附加的，更新或删除是不鼓励或禁止的。事件旨在反映应用程序级别发生的事情，而不是低级状态更改。

事件源是一种强大的数据建模技术：从应用程序的角度来看，将用户的行为记录为不可变的事件更有意义，而不是记录这些行为对可变数据库的影响。事件采购使得随着时间的推移而逐渐发展应用程序变得更加容易，通过更容易理解事情发生的原因以及防范应用程序错误（请参阅“不可变事件的优点”），帮助进行调试。

例如，存储“学生取消课程注册”事件清楚地表达了单一行为的中性意图，而副作用“从注册表中删除了一个条目，并且一个取消原因被添加到学生反馈表“嵌入了很多有关方式的假设数据稍后将被使用。如果引入新的应用程序功能，例如“将地点提供给等待列表中的下一个人”——事件顺序方法允许将新的副作用轻松地链接到现有事件上。

事件顺序类似于编年史数据模型【45】，事件日志和事实表之间也有相似之处，您可以在星型模式中找到它（参阅“星型和雪花型：分析的模式”）。

专门的数据库如Event Store【46】已经被开发来支持使用事件采购的应用程序，但总的来说，这个方法是独立于任何特定的工具的。传统的数据库或基于日志的消息代理也可以用来构建这种风格的应用程序。

从事件日志中导出当前状态

事件日志本身并不是很有用，因为用户通常期望看到系统的当前状态，而不是修改的历史。例如，在购物网站上，用户期望能够看到他们购物车的当前内容，而不是他们对购物车所做的所有改变的附加列表。

因此，使用事件源的应用程序需要记录事件的日志（表示写入系统的数据），并将其转换为适合向用户显示的应用程序状态（从系统读取数据的方式【47】）。这种转换可以使用任意的逻辑，但它应该是确定性的，以便您可以再次运行它并从事件日志中派生相同的应用程序状态。

与更改数据捕获一样，重放事件日志可以让您重新构建系统的当前状态。但是，日志压缩需要以不同的方式处理：

- 用于记录更新的CDC事件通常包含记录的全部新版本，因此主键的当前值完全由该主键的最近事件确定，并且日志压缩可以丢弃以前的事件同样的钥匙。
- 另一方面，事件采购是将事件建模为更高的级别：事件通常表示用户操作的意图，而不是由于操作而发生的状态更新的机制。在这种情况下，以后的事件通常不会覆盖以前的事件，所以您需要事件的完整历史来重构最终状态。日志压缩不可能以相同的方式进行。

使用事件源的应用程序通常有一些机制来存储从事件日志中导出的当前状态的快照，因此它们不需要重复处理完整的日志。但是，这只是一个性能优化，可以加快读取和恢复崩溃的速度;目的是系统能够永久存储所有原始事件，并在需要时重新处理完整的事件日志。我们在第463页的“不变性的限制”中讨论这个假设。

命令和事件

事件采购哲学是仔细区分事件和命令【48】。当来自用户的请求首先到达时，它最初是一个命令：在这一点上它可能仍然失败，例如因为违反了一些完整性条件。应用程序必须首先验证它是否可以执行该命令。如果验证成功并且命令被接受，则它变成一个持久且不可变的事件。

例如，如果用户试图注册特定用户名，或在飞机上或剧院中预定座位，则应用程序需要检查用户名或座位是否已被占用。（我们先前在第364页的“容错概念”中讨论过这个例子。）当检查成功时，应用程序可以生成一个事件来指示特定的用户名是由特定的用户ID注册的，座位已经预留给特定的顾客。

在事件发生的时候，这成为事实。即使客户稍后决定更改或取消预订，事实仍然是事实，他们以前曾为某个特定的座位进行预订，而更改或取消是稍后添加的单独事件。

事件流的消费者不允许拒绝事件：当消费者看到事件时，它已经是日志中不可变的部分，并且可能已经被其他消费者看到。因此，任何命令的验证都需要在事件成为事件之前同步发生，例如，通过使用可自动验证命令并发布事件的可序列化事务。

或者，预订座位的用户请求可以分成两个事件：第一个是暂时预约，第二个是确认预约后的单独确认事件（如第350页上的“使用总订单广播实现线性化存储”中所述）。这个分割允许验证发生在一个异步的过程中。

状态，流和不变性

我们在第10章中看到批量处理从其输入文件的不变性中受益，因此您可以在现有输入文件上运行实验性处理作业，而不用担心损坏它们。这种不变性原则也是使得事件采购和数据变化如此强大的原因。

我们通常将数据库视为存储应用程序的当前状态 - 这种表示法针对读取进行了优化，而且通常对于服务查询来说是最方便的。状态的本质是它的变化，所以数据库支持更新和删除数据以及插入数据。这是如何符合不变性的？

只要你的状态发生了变化，那么这个状态是随着时间的推移而变化的事件的结果。例如，您当前可用的座位列表是您已经处理的预订的结果，当前帐户余额是帐户中的信用卡和借方的结果，而您的Web服务器的响应时间图是发生的所有Web请求的个别响应时间。

无论国家如何变化，总会有一系列事件导致这些变化。即使事情已经解决，事实仍然是事实发生的事实。关键的想法是可变状态和不可变事件的附加日志不相互矛盾：它们是同一枚硬币的两面。所有变化的日志，变化日志，代表了随着时间的推移状态的演变。

如果您有数学上的倾向，那么您可能会说应用程序状态是随着时间的推移整合了一个事件流而得到的，而且当您按照时间区分状态时会得到一个更改流，如图11-6所示【49,50,51】。这个比喻有一定的局限性（例如，国家的二阶导数似乎没有意义），但这是考虑数据的一个有用的起点。

$$state(now) = \int_{t=0}^{now} stream(t)\,\mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\,state(t)}{\mathrm{d}t}$$

图11-6 当前应用程序状态和事件流之间的关系

如果你持久地存储更新日志，那么这只是使状态重现的效果。如果你认为事件的日志是你的记录系统，并且从它派生出任何可变状态，那么就更容易推断通过系统的数据流。正如帕特·赫兰（Pat Helland）所说的【52】：

> 事务日志记录对数据库所做的所有更改。高速追加是更改日志的唯一方法。从这个角度来看，数据库的内容会保存日志中最新记录值的缓存。事实是日志。数据库是日志子集的缓存。该缓存子集恰好是来自日志的每个记录和索引值的最新值。

日志压缩（如“日志压缩”中所述）是一种桥接日志和数据库状态之间区别的方法：它只保留每条记录的最新版本，并丢弃被覆盖的版本。

不可变事件的优点

数据库中的不变性是一个古老的想法。例如，会计师在数个世纪以来一直使用不变性财务簿记。当一笔交易发生时，它被记录在一个仅追加分类帐中，这本质上是描述货币，商品或服务已经转手的事件日志。账目，如损益或资产负债表，是从分类账中的交易中加起来得来的【53】。

如果发生错误，会计师不会删除或更改分类帐中的错误交易 - 而是增加另一笔交易，以补偿错误，例如退还不正确的费用。不正确的交易将永远保留在分类帐中，因为审计原因可能很重要。如果从不正确的分类账导出的错误数字已经公布，那么下一个会计期间的数字就包括一个更正。这个过程在会计中是完全正常的【54】。

尽管这种可审计性在金融系统中尤其重要，但对于不受这种严格管制的许多其他系统也是有益的。如“批处理输出的哲学”（第439页）中所述，如果您意外地部署了将错误数据写入数据库的错误代码，那么如果代码能够破坏性地覆盖数据，恢复将更加困难。通过不可变事件的追加日志，诊断发生的事情和从问题中恢复起来要容易得多。

不可变的事件也捕获比当前状态更多的信息。例如，在购物网站上，顾客可以将物品添加到他们的购物车，然后再将其移除。虽然第二个事件从订单履行角度取消了第一个事件，但为了分析目的，客户正在考虑某个特定项目，但是之后决定采取反对措施。也许他们会选择在未来购买，或者他们找到替代品。这个信息被记录在一个事件日志中，但是当它们从购物车中被删除时，这个信息会丢失在删除项目的数据库中【42】。

从同一事件日志中获取多个视图

而且，通过从不变事件日志中分离可变状态，可以从事件的相同日志中派生出几个不同的面向读取的表示。这就像一个流的多个消费者一样工作（图11-5）：例如，分析数据库Druid使用这种方法从Kafka直接获取【55】，Pista chio是一个分布式的键值存储，使用Kafka作为提交日志【56】，Kafka Connect接收器可以将来自Kafka的数据导出到各种不同的数据库和索引【41】。对于许多其他存储和索引系统（如搜索服务器）来说，类似地从分布式日志中获取输入也是有意义的（请参阅“保持系统同步”）。

从事件日志到数据库有一个明确的转换步骤，可以更容易地随时间推移您的应用程序：如果您想要引入一个以新的方式呈现现有数据的新功能，您可以使用事件日志来构建一个单独的新功能的读取优化视图，并与现有的一起运行

系统而不必修改它们。并行运行旧系统和新系统通常比在现有系统中执行复杂的模式迁移更容易。一旦旧的系统不再需要，你可以简单地关闭它并回收它的资源【47,57】。

如果您不必担心如何查询和访问数据，那么存储数据通常是非常简单的。模式设计，索引和存储引擎的许多复杂性都是希望支持某些查询和访问模式的结果（参见第3章）。出于这个原因，通过将数据写入的形式与读取形式分开，并允许几个不同的读取视图，可以获得很大的灵活性。这个想法有时被称为命令查询责任分离（CQRS）【42,58,59】。

数据库和模式设计的传统方法是基于数据必须以与查询相同的形式写入的谬误。有关正常化和非规范化的争论（参阅"多对一和多对多的关系"），如果可以将数据从写入优化的事件日志转换为读取优化的应用程序状态，则变得基本无关紧要：在读取优化的视图中对数据进行非规范化是完全合理的，因为翻译过程为您提供了一种机制，使其与事件日志保持一致。

在"描述负载"中，我们讨论了推特主页时间表，最近一个特定用户正在关注的人（如邮箱）写的最近发布的推文缓存。这是阅读优化状态的另一个例子：家庭时间表高度变形，因为你的推文在所有跟随你的人的时间线上都是重复的。然而，扇出服务保持这种复制状态与新的推文和新的以下关系保持同步，这保持了复制的可管理性。

并发控制

事件采集和更改数据捕获的最大缺点是事件日志的消费者通常是异步的，所以用户可能会写入日志，然后从日志派生的视图中读取并查找他们的写作还没有反映在读取视图。我们在"读己之写"中讨论了这个问题和潜在的解决方案。

一种解决方案是同步执行读取视图的更新，并将事件附加到日志中。这需要一个事务来将写入操作合并到一个原子单元中，所以要么需要将事件日志和读取视图保存在同一个存储系统中，要么需要跨不同系统的分布式事务。或者，您可以使用在"使用全序广播实现线性化存储"中讨论的方法。

另一方面，从事件日志导出当前状态也简化了并发控制的某些方面。对多个对象事务的需求（参阅"单对象和多对象操作"）源于单个用户操作，需要在多个不同的位置更改数据。通过事件采购，您可以设计一个事件，以便对用户操作进行独立的描述。用户操作只需要在一个地方进行一次写操作，即将事件附加到日志中，这很容易使原子化。

如果事件日志和应用程序状态以相同的方式分区（例如，为分区3中的客户处理事件只需要更新应用程序状态的分区3），则直接的单线程日志消费者不需要并发控制（write-by）构造，它一次只处理一个事件（参阅"真的的串行执行"）。该日志通过在分区中定义事件的串行顺序来消除并发性的不确定性【24】。如果一个事件触及多个状态分区，那么需要做更多的工作，我们将在第12章讨论。

不变性的限制

许多不使用事件源模型的系统依赖于不可变性：各种数据库在内部使用不可变的数据结构或多版本数据来支持时间点快照（参见“索引和快照隔离”）。Git，Mercurial和Fossil等版本控制系统也依靠不可变的数据来保存文件的版本历史记录。

永远保持所有变化的不变的历史在多大程度上是可行的？答案取决于数据集中的流失量。一些工作负载主要是添加数据，很少更新或删除;他们很容易使不变。其他工作负载在较小的数据集上有较高的更新和删除率;在这些情况下，不可改变的历史可能变得过于庞大，碎片化可能成为一个问题，压缩和垃圾收集的表现对于操作的鲁棒性变得至关重要【60,61】。

除了性能方面的原因外，也可能出于管理方面的原因需要删除数据的情况，尽管这些数据都是不可变的。例如，隐私条例可能要求在关闭帐户后删除用户的个人信息，数据保护立法可能要求删除错误的信息，或者可能需要包含敏感信息的意外泄露。

在这种情况下，仅仅在日志中添加另一个事件来指示先前的数据应该被视为删除是不够的 —— 您实际上是想重写历史并假装数据从未写在第一位。例如，Datomic调用这个特性excision 【62】，而Fossil版本控制系统有一个类似的概念叫做shunning 【63】。

真正的删除数据是非常困难的【64】，因为拷贝可以存在于很多地方：例如，存储引擎，文件系统和SSD通常写入一个新的位置，而不是覆盖到位【52】，而备份通常是故意不可改变的防止意外删除或腐败。删除更多的是“使检索数据更难”，而不是“使检索数据不可能”。无论如何，有时您必须尝试，正如我们在“立法和自律”中所看到的。

流处理

到目前为止，本章中我们已经讨论了流的来源（用户活动事件，传感器和写入数据库），我们讨论了流如何传输（通过直接消息传送，通过消息代理和事件日志）。

剩下的就是讨论一下你可以用流做什么 —— 也就是说，你可以处理它。一般来说，有三种选择：

1. 您可以将事件中的数据写入数据库，缓存，搜索索引或类似的存储系统，然后由其他客户端查询。如图11-5所示，这是保持数据库与系统其他部分发生更改同步的好方法 —— 特别是当流消费者是写入数据库的唯一客户端时。写入存储系统的流程相当于我们在“批处理工作流程的输出”页面上讨论的内容。
2. 您可以以某种方式将事件推送给用户，例如通过发送电子邮件警报或推送通知，或通过将事件流式传输到可实时显示的实时仪表板。在这种情况下，人是流的最终消费者。
3. 您可以处理一个或多个输入流以产生一个或多个输出流。数据流可能会经过由几个这样的处理阶段组成的流水线，然后才会输出（选项1或2）。

在本章的其余部分中，我们将讨论选项3：处理流以产生其他派生流。处理这样的流的代码片段被称为操作员或作业。它与我们在第10章中讨论过的Unix进程和MapReduce作业密切相关，数据流的模式是相似的：一个流处理器以只读的方式使用输入流，并将其输出写入一个不同的位置时尚。

流处理器中的分区和并行化模式也非常类似于第10章中介绍的MapReduce和数据流引擎，因此我们不在这里重复这些主题。基本的映射操作（如转换和过滤记录）也是一样的。

批量作业的一个关键区别是流不会结束。这种差别有很多含义：正如本章开始部分所讨论的，排序对无界数据集没有意义，因此不能使用排序合并联接（请参阅“减少连接和分组”）。容错机制也必须改变：对于已经运行了几分钟的批处理作业，可以简单地从头开始重新启动失败的任务，但是对于已经运行数年的流作业，在开始后重新开始崩溃可能不是一个可行的选择。

流处理的应用

长期以来，流处理一直用于监控目的，如果某个事情发生，组织就希望得到警报。例如：

- 欺诈检测系统需要确定信用卡的使用模式是否意外地发生了变化，并且如果信用卡可能已被盗用，则将其封锁。
- 交易系统需要检查金融市场的价格变化，并根据指定的规则进行交易。
- 制造系统需要监控工厂中机器的状态，如果出现故障，可以快速识别问题。
- 军事和情报系统需要跟踪潜在的攻击者的行动，并在发生袭击的迹象时发出警报。

这些类型的应用程序需要非常复杂的模式匹配和相关性。然而，流处理的其他用途也随着时间的推移而出现。在本节中，我们将简要比较一下这些应用程序。

复杂的事件处理

复杂事件处理（CEP）是20世纪90年代为分析事件流而开发的一种方法，尤其适用于需要搜索某些事件模式的应用程序【65,66】。与正则表达式允许您在字符串中搜索特定字符模式的方式类似，CEP允许您指定规则以在流中搜索某些事件模式。

CEP系统通常使用高级声明式查询语言（如SQL或图形用户界面）来描述应该检测到的事件模式。这些查询被提交给一个处理引擎，该引擎使用输入流并在内部维护一个执行所需匹配的状态机。当发现匹配时，引擎发出一个复杂的事件（因此名字）与事件模式的细节【67】。

在这些系统中，查询和数据之间的关系与普通数据库相比是颠倒的。通常情况下，数据库会持久存储数据，并将查询视为暂时的：当查询进入时，数据库搜索与查询匹配的数据，然后在查询完成时忘记查询。 CEP引擎反转了这些角色：查询是长期存储的，来自输入流的事件不断流过它们，以搜索匹配事件模式的查询【68】。

CEP的实现包括Esper 【69】，IBM InfoSphere Streams 【70】，Apama，TIBCO StreamBase和SQLstream。像Samza这样的分布式流处理器也获得了对流声明式查询的SQL支持【71】。

流分析

使用流处理的另一个领域是对流进行分析。 CEP和流分析之间的边界是模糊的，但作为一般规则，分析往往不太关心找到特定的事件序列，并且更倾向于聚合和统计度量大量的事件——例如：

- 测量某种类型事件的速率（每个时间间隔发生的频率）
- 计算一段时间内某个值的滚动平均值
- 将当前的统计数据与以前的时间间隔进行比较（例如，检测趋势或提醒与上周同期相比过高或过低的指标）

这些统计信息通常是在固定的时间间隔内进行计算的，例如，您可能想知道在过去5分钟内每秒对服务的平均查询次数，以及在此期间的第99百分位响应时间。在几分钟内平均，从一秒钟到下一秒钟平滑无关的波动，同时还能及时了解交通模式的任何变化。您汇总的时间间隔称为窗口，我们将在“关于时间的推理”中更详细地讨论窗口。

流分析系统有时使用概率算法，例如Bloom filter（我们在“性能优化”中遇到过），设置成员资格，HyperLogLog 【72】基数估计以及各种百分比估计算法（请参阅“实践中的百分位点“第16页）。概率算法产生近似的结果，但是具有在流处理器中比精确算法需要少得多的存储器的优点。近似算法的使用有时会使人们相信流处理系统总是有损和不精确的，但这是错误的：流处理没有任何内在的近似，而概率算法只是一个优化【73】。

许多开源分布式流处理框架的设计都是以分析为基础的：例如Apache Storm，Spark Streaming，Flink，Concord，Samza和Kafka Streams 【74】。托管服务包括Google Cloud Dataflow和Azure Stream Analytics。

保持物化视图

我们在“数据库和数据流”中看到，可以使用数据库更改流来保持派生数据系统（如缓存，搜索索引和数据仓库）与源数据库保持最新。我们可以将这些示例视为维护实体化视图的具体情况（请参阅“聚合：数据立方体和物化视图”）：导出某个数据集的替代视图，以便可以高效地查询它，并在底层数据更改【50】。

同样，在事件采购中，应用程序状态通过应用事件日志来维护;这里的应用状态也是一种物化视图。与流分析场景不同，在某个时间窗口内仅考虑事件通常是不够的：构建物化视图可能需要在任意时间段内的所有事件，除了可能由日志压缩丢弃的任何过时事件（请参阅“日志压缩“）。实际上，您需要一个可以一直延伸到一开始的窗口。

原则上，任何流处理器都可以用于物化视图维护，尽管永久维护事件的需要与一些主要在有限持续时间的窗口上运行的面向分析的框架的假设背道而驰。 Samza和Kafka Streams支持这种用法，建立在Kafka对夯实的支持上【75】。

在流上搜索

除了允许搜索由多个事件组成的模式的CEP外，还有时需要基于复杂的标准（例如全文搜索查询）来搜索单个事件。

例如，媒体监测服务可以订阅新闻文章和媒体广播，并搜索任何关于公司，产品或感兴趣的话题的新闻。这是通过预先制定一个搜索查询来完成的，然后不断地将新闻项目流与这个查询进行匹配。在一些网站上也有类似的功能：例如，房地产网站的用户在市场上出现符合其搜索条件的新房产时，可以要求通知。 Elasticsearch 【76】的渗滤器功能是实现这种流式搜索的一种选择。

传统的搜索引擎首先索引文件，然后在索引上运行查询。相比之下，搜索一个数据流将会处理它的头部：查询被存储，文档通过查询运行，就像CEP一样。在最简单的情况下，您可以针对每个查询来测试每个文档，但是如果您有大量查询，这可能会变慢。为了优化过程，可以对查询和文档进行索引，从而缩小可能匹配的查询集合【77】。

消息传递和RPC

在第136页的“消息传递数据流”中，我们讨论了消息传递系统作为RPC的替代方案，即作为通信服务的机制，例如在参与者模型中所使用的。虽然这些系统也是基于消息和事件，但我们通常不会将它们视为流处理器：

Actor框架主要是管理通信模块的并发和分布式执行的机制，而流处理主要是数据管理技术。

- 参与者之间的交流往往是短暂的，而且是一对一的，而事件日志则是持久的，多用户的。
- 参与者可以以任意方式进行通信（包括循环请求/响应模式），但流处理器通常设置在非循环流水线中，其中每个流是一个特定作业的输出，并且从一组明确定义的输入流派生。

也就是说，RPC类系统和流处理之间有一些交叉区域。例如，Apache Storm有一个称为分布式RPC的功能，它允许将用户查询分散到一系列也处理事件流的节点上;这些查询然后与来自输入流的事件交织，结果可以被汇总并发回给用户【78】。（另参阅“多分区数据处理”）

也可以使用actor框架来处理流。但是，很多这样的框架在崩溃的情况下不能保证消息的传递，所以这个过程不是容错的，除非你实现了额外的重试逻辑。

关于时间的推理

流处理器通常需要处理时间，特别是在用于分析目的的时候，频繁使用时间窗口，例如“过去五分钟的平均时间”。“最后五分钟”的含义似乎应该是未知的，大而清晰，但不幸的是这个概念是令人惊讶的棘手。

在批处理过程中，处理任务通过大量的历史事件迅速收缩。如果需要按时间分类，批处理需要查看每个事件中嵌入的时间戳。查看运行批处理的机器的系统时钟没有意义，因为处理运行的时间与事件实际发生的时间无关。

批处理可以在几分钟内读取一年的历史事件;在大多数情况下，感兴趣的时间表是历史的一年，而不是几分钟的处理。而且，在事件中使用时间戳允许处理确定性的：在相同的输入上再次运行相同的处理过程会得到相同的结果（参阅“故障容错”）。

另一方面，许多流处理框架使用处理机器上的本地系统时钟（处理时间）来确定窗口【79】。这种方法具有简单的优点，事件创建和事件处理之间的延迟可以忽略不计。然而，如果存在任何显着的处理滞后，即处理可能比事件实际发生的时间显着晚，则会中断处理。

事件时间与处理时间

有许多原因可能会延迟处理：排队，网络故障（请参阅第267页的“不可靠的网络”），导致消息代理或处理器中出现争用的性能问题，重新启动流消费者或重新处理过去的事件（参阅“重播旧消息”），或者在修复代码中的错误之后进行恢复。

而且，消息延迟还可能导致消息的不可预知的排序。例如，假设用户首先发出一个Web请求（由Web服务器A处理），然后发出第二个请求（由服务器B处理）。 A和B发出描述他们处理的请求的事件，但是B的事件在A的事件发生之前到达消息代理。现在，流处理器将首先看到B事件，然后看到A事件，即使它们实际上是以相反的顺序发生的。

如果有一个类比的话，可以考虑一下“星球大战”的电影：第四集于1977年发行，1980年第五集，1983年第六集，之后分别在1999年，2002年和2005年发行第一，二，三集，以及2015年的第七集【80】[ii]。如果你按照他们出来的顺序观看电影，你处理电影的顺序与他们叙述的顺序是不一致的。（情节编号就像事件时间戳一样，观看电影的日期就是处理时间。）作为人类，我们能够应对这种不连续性，但是流处理算法需要专门编写以适应这种情况时间安排和订购问题。

> ii. 感谢Flink社区的Kostas Kloudas提出这个比喻。 ↩

令人困惑的事件时间和处理时间导致错误的数据。例如，假设您有一个流处理器来测量请求率（计算每秒请求数）。如果您重新部署流处理器，则可能会关闭一分钟，并在事件恢复时处理积压的事件。如果您根据处理时间来衡量速率，那么看起来好像在处理积压时突然出现异常的请求高峰，而事实上请求的实际速率是稳定的（图11-7）。

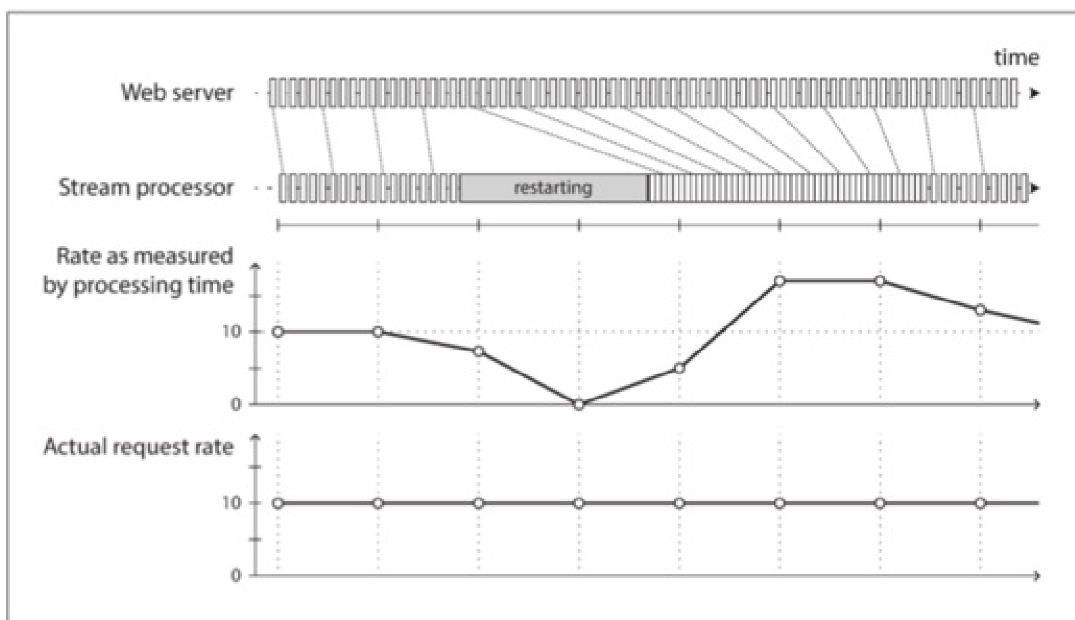


图11-7 由于处理时间的窗口化，由于处理速率的变化而引入人为因素

知道什么时候你准备好了

从事件时间的角度来定义窗口时，一个棘手的问题是，当你收到特定窗口的所有事件，或者是否还有事件发生时，你永远无法确定。

例如，假设您将事件分组为一分钟的窗口，以便您可以统计每分钟的请求数。你已经计算了一些事件，这些事件的时间戳是在第37分钟的时间落下的，时间已经推移了。现在大部分的事件都在一小时的第38和第39分钟之内。你什么时候宣布你已经完成了第37分钟的窗口，并输出其计数器值？

在一段时间没有看到任何新的事件之后，您可以超时并宣布一个窗口，但仍然可能发生某些事件被缓存在另一台计算机上，由于网络中断而延迟。您需要能够处理窗口已经声明完成后到达的这样的滞留事件。大体上，你有两个选择【1】：

1. 忽略这些零散的事件，因为它们在正常情况下可能只是一小部分事件。您可以将丢弃事件的数量作为度量标准进行跟踪，并在您开始丢弃大量数据时发出警报。
2. 发布一个更正，更新的窗口与包含散兵队员的价值。您可能还需要收回以前的输出。

在某些情况下，可以使用特殊的消息来指示“从现在开始，不会有比t更早的时间戳的消息”，消费者可以使用它来触发窗口【81】。但是，如果不同机器上的多个生产者正在生成事件，每个事件都有自己的最小时间戳阈值，则消费者需要分别跟踪每个生产者。在这种情况下添加和删除生产者是比较棘手的。

你用的是什么时间？

当事件可以在系统中的多个点缓冲时，为事件分配时间戳更加困难。例如，考虑将使用率度量的事件报告给服务器的移动应用程序。该应用程序可能会在设备处于脱机状态时使用，在这种情况下，它将在设备上本地缓冲事件，并在下一次可用的互联网连接（可能是几小时甚

至几天）时将它们发送到服务器。对于这个流的任何消费者来说，这些事件将显示为极其滞后的落后者。

在这种情况下，根据移动设备的本地时钟，事件的时间戳实际上应该是发生用户交互的时间。但是，用户控制的设备上的时钟通常是不可信的，因为它可能会被意外或故意设置为错误的时间（请参见“时钟同步与准确性”）。服务器收到事件的时间（根据服务器的时钟）更可能是准确的，因为服务器在您的控制之下，但在描述用户交互方面意义不大。

要调整不正确的设备时钟，一种方法是记录三个时间戳【82】：

- 事件发生的时间，根据设备时钟
- 根据设备时钟将事件发送到服务器的时间
- 服务器根据服务器时钟收到事件的时间

通过从第三个时间戳中减去第二个时间戳，可以估算设备时钟和服务器时钟之间的偏移量（假设网络延迟与所需的时间戳精度相比可忽略不计）。然后，您可以将该偏移量应用于事件时间戳，从而估计事件实际发生的真实时间（假设设备时钟偏移在事件发生的时间与发送到服务器的时间之间没有变化）。

这个问题对于流处理来说并不是唯一的，批处理遇到了与时间推理完全相同的问题。在一个流式环境中，我们更加注意到时间的流逝。

窗口的类型

一旦你知道如何确定一个事件的时间戳，下一步就是决定如何定义一段时间的窗口。窗口然后可以用于聚合，例如计数事件，或计算窗口内的值的平均值。有几种窗口是常用的【79,83】：

***Tumbling*窗口**

一个翻滚的窗口有一个固定的长度，每个事件都属于一个窗口。例如，如果您有1分钟的翻滚窗口，则所有时间戳在 `10:03:00` 和 `10:03:59` 之间的事件会被分组到一个窗口中， `10:04:00` 和 `10:04:59` 之间的事件下一个窗口，等等。您可以通过获取每个事件时间戳并将其四舍五入到最接近的分钟来确定它所属的窗口，从而实现1分钟的翻滚窗口。

***Hopping*窗**

跳频窗口也具有固定的长度，但允许窗口重叠以提供一些平滑。例如，1分钟跳跃大小的5分钟窗口将包含 `10:03:00` 至 `10:07:59` 之间的事件，则下一个窗口将覆盖 `10:04:00` 至 `10:08` 之间的事件： 59，等等。您可以通过首先计算1分钟滚动窗口，然后聚合在几个相邻的窗口上来实现此跳频窗口。

滑动窗口

滑动窗口包含在彼此的某个间隔内发生的所有事件。例如，一个5分钟的滑动窗口将覆盖10点03分39秒和10点08分12秒的事件，因为它们相距不到5分钟（注意翻滚和跳跃的5分钟窗口不会把这两个事件在同一个窗口中，因为他们使用固定的边界）。滑动窗口可以通过保持按时间排序的事件缓冲区并在从窗口到期时移除旧事件来实现。

会话窗口

与其他窗口类型不同，会话窗口没有固定的持续时间。相反，它是通过将同一用户的所有事件分组在一起，并在时间上紧密地组合在一起来定义的，并且当用户在一段时间内不活动时（例如，如果30分钟内没有事件），窗口结束。会话化是网站分析的常见要求（参阅“GROUP BY”）。

流式连接

在第10章中，我们讨论了批处理作业如何通过关键连接数据集，以及这种连接如何构成数据管道的重要组成部分。由于流处理将数据管道概括为对无界数据集进行增量处理，因此对流进行连接的需求也完全相同。

然而，新事件随时可能出现在一个流中，这使得加入流比批处理作业更具挑战性。为了更好地理解情况，我们来区分三种不同类型的连接：流-流连接，流表连接和表连接【84】。在下面的章节中，我们将通过例子来说明。 流 - 流连接（窗口连接）

假设您的网站上有搜索功能，并且想要检测搜索到的网址的近期趋势。每次有人输入搜索查询时，都会记录包含查询和返回结果的事件。每当有人点击其中一个搜索结果时，就会记录另一个记录点击的事件。为了计算搜索结果中每个网址的点击率，您需要将搜索操作和点击操作的事件组合在一起，这些事件通过具有相同的会话ID进行连接。广告系统需要类似的分析【85】。

如果用户放弃他们的搜索，点击可能永远不会到来，即使它到了，搜索和点击之间的时间可能是高度可变的：在很多情况下，它可能是几秒钟，但可能长达几天或几周（如果用户运行搜索，忘记关于该浏览器选项卡，然后返回到选项卡，稍后再单击一个结果）。由于可变的网络延迟，点击事件甚至可能在搜索事件之前到达。您可以选择合适的加入窗口，例如，如果间隔至多一小时发生一次搜索，您可以选择加入搜索。

请注意，在click事件中嵌入搜索的细节并不等同于加入事件：这样做只会告诉您有关用户单击搜索结果的情况，而不是用户未点击任何搜索结果的搜索结果。为了衡量搜索质量，您需要准确的点击率，为此您需要搜索事件和点击事件。

为了实现这种类型的连接，流处理器需要维护状态：例如，在最后一小时发生的所有事件，都由会话标识索引。无论何时发生搜索事件或点击事件，都会将其添加到适当的索引，并且流处理器还检查另一个索引，以查看是否已经到达同一会话ID的另一个事件。如果有匹配的事件，则发出说明哪个搜索结果被点击的事件。如果搜索事件过期而没有看到匹配的点击事件，则会发出说明哪些搜索结果未被点击的事件。

流表连接（stream enrichment）

在“示例：用户活动事件分析”（图10-2）中，我们看到了加入两个数据集的批量作业示例：一组用户活动事件和一个用户配置文件数据库。将用户活动事件视为流，并在流处理器中连续执行相同的连接是很自然的：输入是包含用户ID的活动事件流，并且输出是活动事件流，其中用户ID已经用关于用户的简档信息来扩充。这个过程有时被称为使用来自数据库的信息来丰富活动事件。

要执行此联接，流程过程需要一次查看一个活动事件，在数据库中查找事件的用户标识，并将该概要信息添加到活动事件中。数据库查询可以通过查询远程数据库来实现;但是，正如在“示例：分析用户活动事件”一节中讨论的，此类远程查询可能会很慢并且有可能导致数据库过载【75】。

另一种方法是将数据库副本加载到流处理器中，以便在本地进行查询而无需网络往返。这种技术与我们在“Map端连接”中讨论的散列连接非常相似：如果数据库的本地副本足够小，则本地副本可能是内存中的散列表，或者是本地磁盘上的索引。

与批处理作业的区别在于批处理作业使用数据库的时间点快照作为输入，而流处理器是长时间运行的，并且数据库的内容可能随时间而改变，所以流处理器数据库的本地副本需要保持最新。这个问题可以通过更改数据捕获来解决：流处理器可以订阅用户配置文件数据库的更新日志以及活动事件流。在创建或修改配置文件时，流处理器会更新其本地副本。因此，我们获得两个流之间的连接：活动事件和配置文件更新。

流表连接实际上非常类似于流 - 流连接;最大的区别在于对于表changelog流，连接使用一个可以回溯到“开始时间”（概念上是无限的窗口）的窗口，新版本的记录会覆盖较早的版本。对于流输入，连接可能根本没有维护窗口。

表格表连接（物化视图维护）

考虑我们在“描述负载”中讨论的推特时间线例子。我们说过，当用户想要查看他们的主页时间线时，对用户所关注的所有人进行迭代是非常昂贵的，推文，并合并它们。

相反，我们需要一个时间线缓存：一种每个用户的“收件箱”，在发送推文的时候写入这些信息，以便读取时间线是一次查询。实现和维护此缓存需要以下事件处理：

- 当用户发送新的推文时，它将被添加到每个跟随你的用户的时间线上。
- 用户删除推文时，将从所有用户的时间表中删除。
- 当用户u1开始跟随用户u2时，u2最近的tweets被添加到u1的时间线上。
- 当用户u1取消关注用户u2时，u1的推文将从u1的时间线中移除。

要在流处理器中实现这种缓存维护，需要用于推文（发送和删除）和跟随关系（跟随和取消跟随）的事件流。流过程需要维护一个包含每个用户关注者集合的数据库，以便知道当一个新的tweet到达时需要更新哪些时间轴【86】。 查看这个流过程的另一种方式是维护一个连接两个表（tweet和follow）的查询的物化视图，如下所示：

```
SELECT follows.follower_id AS timeline_id,
    array_agg(tweets.* ORDER BY tweets.timestamp DESC)
FROM tweets
JOIN follows ON follows.followee_id = tweets.sender_id
GROUP BY follows.follower_id
```

流的连接直接对应于该查询中的表的连接。时间轴实际上是这个查询结果的缓存，每当基础表发生变化时都会更新[iii]。

> iii. 如果你把一个流视为一个表的衍生物，如图11-6所示，并且把一个连接看作是两个表u·v的乘积，那么会发生一些有趣的事情：物化连接的变化流遵循产品规则(u·v)'= u'v + uv'(u·v)'= u'v + uv'。 换句话说，任何tweets的变化都与当前的追随者联系在一起，任何追随者的变化都与当前的tweets【49,50】相结合。↩

连接的时间依赖性

这就产生了一个问题：如果不同的事件发生在相似的时间周围，他们按照何种顺序进行处理？在流表连接示例中，如果用户更新其配置文件，哪些活动事件与旧配置文件（在配置文件更新之前处理）结合，哪些与新配置文件结合（在配置文件更新之后处理）？换句话说：如果状态随着时间的推移而改变，并且你加入了某个状态，那么你使用什么时间点来加入【45】？

这种时间依赖性可能发生在许多地方。例如，如果您销售东西，则需要对发票进行适当的税率，这取决于国家或州，产品类型和销售日期（因为税率会随时变化）。将销售额加入税率表时，如果您正在重新处理历史数据，您可能希望加入销售时的税率，这可能与当前的税率不同。

如果跨流的事件排序是未确定的，那么这个连接变得不确定【87】，这意味着你不能在相同的输入上重新运行相同的工作，并且必然会得到相同的结果：输入流上的事件可能交织在当你再次运行这个工作时，采用不同的方式

在数据仓库中，这个问题被称为缓慢变化的维度（SCD），通常通过对特定版本的联合记录使用唯一的标识符来解决：例如，每当税率改变时，新的标识符，并且发票包括销售时的税率标识符【88,89】。这种变化使连接成为确定性的，但是由于表中所有记录的版本都需要保留，导致日志压缩是不可能的。

容错

在本章的最后一节中，让我们考虑流处理器如何容忍错误。我们在第10章中看到，批处理框架可以很容易地容忍错误：如果MapReduce作业中的任务失败，可以简单地在另一台机器上重新启动，并且丢弃失败任务的输出。这种透明的重试是可能的，因为输入文件是不可变的，每个任务都将其输出写入到HDFS上的单独文件，并且输出仅在任务成功完成时可见。

特别是，批处理容错方法可确保批处理作业的输出与没有出错的情况相同，即使事实上某些任务失败了。看起来好像每个输入记录都被处理了一次 —— 没有记录被跳过，而且没有处理两次。尽管重新启动任务意味着实际上可能会多次处理记录，但输出中的可见效果好像只处理过一次。这个原则被称为一次语义学，虽然有效 —— 一次将是一个更具描述性的术语【90】。

在流处理过程中也出现了同样的容错问题，但是处理起来不那么直观：等到某个任务完成之后才使其输出可见，因为流是无限的，因此您永远无法完成处理。

小批量和检查点

一个解决方案是将流分解成小块，并像小型批处理一样处理每个块。这种方法被称为小批量（**microbatching**），它被用于Spark Streaming 【91】。批处理大小通常约为1秒，这是性能折中的结果：较小的批次会导致更大的调度和协调开销，而较大的批次意味着流处理器的结果变得可见之前的较长延迟。

微缩也隐含地提供了与批量大小相等的翻滚窗口（通过处理时间而不是事件时间戳）。任何需要更大窗口的作业都需要明确地将状态从一个微阵列转移到下一个微阵列。

Apache Flink中使用的一种变体方法是定期生成状态滚动检查点并将其写入持久存储器【92,93】。如果流操作符崩溃，它可以从最近的检查点重新启动，并放弃在最后一个检查点和崩溃之间生成的任何输出。检查点由消息流中的条形码触发，类似于微型图形之间的边界，但不强制特定的窗口大小。

在流处理框架的范围内，微观网格化和检查点方法提供了与批处理一样的一次语义。但是，只要输出离开流处理器（例如，通过写入数据库，向外部消息代理发送消息或发送电子邮件），框架将不再能够放弃失败批处理的输出。在这种情况下，重新启动失败的任务会导致外部副作用发生两次，单独使用微配量或检查点不足以防止此问题。

原子提交重访

为了在出现故障时给出精确的一次处理，我们需要确保处理事件的所有输出和副作用只有当处理成功时才会生效。这些影响包括发送给下游运营商或外部消息传递系统（包括电子邮件或推送通知）的任何消息，任何数据库写入，对运营商状态的任何更改以及对输入消息的任何确认（包括将消费者偏移量向前移动基于日志的消息代理）。

这些事情要么都是原子地发生，要么都不发生，但是不应该彼此不同步。如果这种方法听起来很熟悉，那是因为我们在分布式事务和两阶段提交的情况下，在第360页的“准确一次的消息处理”中讨论了它。

在第9章中，我们讨论了分布式交易（如XA）的传统实现中的问题。然而，在更受限制的环境中，可以有效地实现这样的原子提交设施。 Google云数据流【81,92】和VoltDB 【94】中使用了这种方法，并计划在Apache Kafka 【95,96】中添加类似的功能。与XA不同，这些实现

不会尝试跨异构技术提供事务，而是通过在流处理框架中管理状态更改和消息传递来保持内部事务。事务协议的开销可以通过在单个事务中处理几个输入消息来分摊。

幂等

我们的目标是放弃任何失败的任务的部分输出，以便他们可以安全地重试，而不会两次生效。分布式事务是实现这一目标的一种方式，但另一种方式是依赖幂等性【97】。

幂等操作是可以多次执行的操作，并且与只执行一次操作具有相同的效果。例如，将键值存储中的某个键设置为某个固定值是幂等的（再次写入该值会覆盖具有相同值的值），而递增计数器不是幂等的（再次执行递增意味着该值递增两次）。

即使一个操作不是天生的幂等，它往往可以与一些额外的元数据幂等。例如，在使用来自卡夫卡的消息时，每条消息都有一个持续的，单调递增的偏移量。将值写入外部数据库时，可以将触发上次写入的消息的偏移量与值包含在一起。因此，您可以判断是否已应用更新，并避免再次执行相同的更新。

风暴三叉戟的状态处理基于类似的想法【78】。依赖幂等性意味着一些假设：重启一个失败的任务必须以相同的顺序重播相同的消息（一个基于日志的消息代理这样做），处理必须是确定性的，其他节点不能同时更新相同的值【98,99】。

当从一个处理节点故障转移到另一个处理节点时，可能需要进行防护（参阅“领导和锁”），以防止被认为是死的节点的干扰

失败后重建状态

任何需要状态的流进程（例如，任何窗口聚合（例如计数器，平均值和直方图）以及用于连接的任何表和索引）都必须确保在失败之后可以恢复此状态。

一种选择是将状态保持在远程数据存储中并复制它，尽管如每个单独消息的远程数据库查询速度可能会很慢，正如在“流表连接”中所述。另一种方法是保持流处理器的本地状态，并定期复制。然后，当流处理器从故障中恢复时，新任务可以读取复制状态并恢复处理而不丢失数据。

例如，Flink定期捕获操作员状态的快照，并将它们写入HDFS等持久存储器中【92,93】。Samza和Kafka Streams通过将状态更改发送到具有日志压缩功能的专用Kafka主题来复制状态更改，这类似于更改数据捕获【84,100】。VoltDB通过冗余处理多个节点上的每个输入消息来复制状态（参阅“眞的串行执行”）。

在某些情况下，甚至可能不需要复制状态，因为它可以从输入流重建。例如，如果状态由一个相当短的窗口中的聚合组成，则它可能足够快，以便重放与该窗口相对应的输入事件。如果状态是通过更改数据捕获维护的数据库的本地副本，那么也可以从日志压缩的更改流重建数据库（请参阅“日志压缩”一节）。

但是，所有这些权衡取决于底层基础架构的性能特征：在某些系统中，网络延迟可能低于磁盘访问延迟，网络带宽可能与磁盘带宽相当。在所有情况下都没有普遍理想的权衡，随着存储和网络技术的发展，本地和远程状态的优点也可能会发生变化。

本章小结

在本章中，我们讨论了事件流，他们所服务的目的以及如何处理它们。在某些方面，流处理非常类似于我们在第10章讨论的批处理，而是在无限的（永无止境的）流而不是固定大小的输入上持续进行。从这个角度来看，消息代理和事件日志可以作为文件系统的流媒体。

我们花了一些时间比较两种消息代理：

AMQP/JMS风格的消息代理

经纪人将个人消息分配给消费者，消费者在成功处理个人消息时确认消息。消息被确认后从代理中删除。这种方法适合作为RPC的异步形式（另请参阅“消息传递数据流”），例如在任务队列中，消息处理的确切顺序并不重要，没有在处理之后，需要重新读取旧消息。

基于日志的消息代理

代理将分区中的所有消息分配给相同的使用者节点，并始终以相同的顺序传递消息。并行性是通过划分来实现的，消费者通过检查他们所处理的最后一个消息的偏移来跟踪他们的进度。代理将消息保留在磁盘上，因此如有必要，可以跳回并重新读取旧消息。

基于日志的方法与数据库中的复制日志（参见第5章）和日志结构存储引擎（请参阅第3章）具有相似之处。我们看到，这种方法特别适用于消耗输入流并生成派生状态或派生输出流的流处理系统。

就流的来源而言，我们讨论了几种可能性：用户活动事件，提供定期读数的传感器和数据馈送（例如金融市场数据）自然地表示为流。我们看到，将数据写入数据流也是有用的：我们可以捕获更改日志 —— 即对数据库所做的所有更改的历史记录 —— 隐式地通过更改数据捕获或通过事件明确地捕获采购。日志压缩允许流保留数据库内容的完整副本。

将数据库表示为流为系统集成提供了强大的机会。您可以通过使用更改日志并将其应用于派生系统，使派生的数据系统（如搜索索引，缓存和分析系统）保持最新。您甚至可以从头开始，从开始一直到现在消耗更改的日志，从而为现有数据构建新的视图。

将状态保持为流并重放消息的设施也是在各种流处理框架中实现流连接和容错的技术的基础。我们讨论了流处理的几个目的，包括搜索事件模式（复杂事件处理），计算加窗聚合（流分析）以及保持派生数据系统处于最新状态（物化视图）。

然后我们讨论了在流处理器中推理时间的困难，包括处理时间和事件时间戳之间的区别，以及在你认为窗口完成之后处理到达的离散事件的问题。

我们区分了可能出现在流程中的三种类型的连接：

流式流连接

两个输入流由活动事件组成，并且连接操作符搜索在某个时间窗口内发生的相关事件。例如，它可以匹配相同用户在30分钟内采取的两个动作。如果您想要在一个流中查找相关事件，则两个连接输入实际上可以是相同的流（自连接）。

流表连接

一个输入流由活动事件组成，另一个输入流是数据库更改日志。更新日志保持数据库的本地副本最新。对于每个活动事件，连接运算符将查询数据库并输出一个丰富的活动事件。

表格连接

两个输入流都是数据库更新日志。在这种情况下，一方的每一个变化都与另一方的最新状态相结合。结果是对两个表之间的连接的物化视图进行了一系列更改。

最后，我们讨论了在流处理器中实现容错和一次语义的技术。与批处理一样，我们需要放弃任何失败任务的部分输出。然而，由于流程长时间运行并持续产生输出，所以不能简单地丢弃所有的输出。相反，可以使用更细粒度的恢复机制，基于微博，检查点，事务或幂等写入。

参考文献

1. Tyler Akidau, Robert Bradshaw, Craig Chambers, et al.: “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing,” *Proceedings of the VLDB Endowment*, volume 8, number 12, pages 1792–1803, August 2015. doi:10.14778/2824032.2824076
2. Harold Abelson, Gerald Jay Sussman, and Julie Sussman: *Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press, 1996. ISBN: 978-0-262-51087-5, available online at *mitpress.mit.edu*
3. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec: “The Many Faces of Publish/Subscribe,” *ACM Computing Surveys*, volume 35, number 2, pages 114–131, June 2003. doi:10.1145/857076.857078
4. Joseph M. Hellerstein and Michael Stonebraker: *Readings in Database Systems*, 4th edition. MIT Press, 2005. ISBN: 978-0-262-69314-1, available online at *redbook.cs.berkeley.edu*
5. Don Carney, Uğur Çetintemel, Mitch Cherniack, et al.: “Monitoring Streams – A New Class of Data Management Applications,” at *28th International Conference on Very Large Data Bases* (VLDB), August 2002.
6. Matthew Sackman: “Pushing Back,” *lshift.net*, May 5, 2016.

7. Vicent Martí: “Brubeck, a statsd-Compatible Metrics Aggregator,” *githubengineering.com*, June 15, 2015.
8. Seth Lowenberger: “MoldUDP64 Protocol Specification V 1.00,” *nasdaqtrader.com*, July 2009.
9. Pieter Hintjens: *ZeroMQ – The Guide*. O'Reilly Media, 2013. ISBN: 978-1-449-33404-8
10. Ian Malpass: “Measure Anything, Measure Everything,” *codeascraft.com*, February 15, 2011.
11. Dieter Plaetinck: “25 Graphite, Grafana and statsd Gotchas,” *blog.raintank.io*, March 3, 2016.
12. Jeff Lindsay: “Web Hooks to Revolutionize the Web,” *progrium.com*, May 3, 2007.
13. Jim N. Gray: “Queues Are Databases,” Microsoft Research Technical Report MSR-TR-95-56, December 1995.
14. Mark Hapner, Rich Burridge, Rahul Sharma, et al.: “JSR-343 Java Message Service (JMS) 2.0 Specification,” *jms-spec.java.net*, March 2013.
15. Sanjay Aiyagari, Matthew Arrott, Mark Atwell, et al.: “AMQP: Advanced Message Queuing Protocol Specification,” Version 0-9-1, November 2008.
16. “Google Cloud Pub/Sub: A Google-Scale Messaging Service,” *cloud.google.com*, 2016.
17. “Apache Kafka 0.9 Documentation,” *kafka.apache.org*, November 2015.
18. Jay Kreps, Neha Narkhede, and Jun Rao: “Kafka: A Distributed Messaging System for Log Processing,” at *6th International Workshop on Networking Meets Databases* (NetDB), June 2011.
19. “Amazon Kinesis Streams Developer Guide,” *docs.aws.amazon.com*, April 2016.
20. Leigh Stewart and Sijie Guo: “Building DistributedLog: Twitter’s High-Performance Replicated Log Service,” *blog.twitter.com*, September 16, 2015.
21. “DistributedLog Documentation,” Twitter, Inc., *distributedlog.io*, May 2016.
22. Jay Kreps: “Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines),” *engineering.linkedin.com*, April 27, 2014.
23. Kartik Paramasivam: “How We’re Improving and Advancing Kafka at LinkedIn,” *engineering.linkedin.com*, September 2, 2015.
24. Jay Kreps: “The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction,” *engineering.linkedin.com*, December 16, 2013.

25. Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *3rd ACM Symposium on Cloud Computing* (SoCC), October 2012.

26. Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: "Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

27. P. P. S. Narayan: "Sherpa Update," *developer.yahoo.com*, June 8, .

28. Martin Kleppmann: "Bottled Water: Real-Time Integration of PostgreSQL and Kafka," *martin.kleppmann.com*, April 23, 2015.

29. Ben Osheroff: "Introducing Maxwell, a mysql-to-kafka Binlog Processor," *developer.zendesk.com*, August 20, 2015.

30. Randall Hauch: "Debezium 0.2.1 Released," *debezium.io*, June 10, 2016.

31. Prem Santosh Udaya Shankar: "Streaming MySQL Tables in Real-Time to Kafka," *engineeringblog.yelp.com*, August 1, 2016.

32. "Mongoriver," Stripe, Inc., *github.com*, September 2014.

33. Dan Harvey: "Change Data Capture with Mongo + Kafka," at *Hadoop Users Group UK*, August 2015.

34. "Oracle GoldenGate 12c: Real-Time Access to Real-Time Information," Oracle White Paper, March 2015.

35. "Oracle GoldenGate Fundamentals: How Oracle GoldenGate Works," Oracle Corporation, *youtube.com*, November 2012.

36. Slava Akhmechet: "Advancing the Realtime Web," *rethinkdb.com*, January 27, 2015.

37. "Firebase Realtime Database Documentation," Google, Inc., *firebase.google.com*, May 2016.

38. "Apache CouchDB 1.6 Documentation," *docs.couchdb.org*, 2014.

39. Matt DeBergalis: "Meteor 0.7.0: Scalable Database Queries Using MongoDB Oplog Instead of Poll-and-Diff," *info.meteor.com*, December 17, 2013.

40. "Chapter 15. Importing and Exporting Live Data," VoltDB 6.4 User Manual, *docs.voltdb.com*, June 2016.

41. Neha Narkhede: "Announcing Kafka Connect: Building Large-Scale Low-Latency Data Pipelines," *confluent.io*, February 18, 2016.

42. Greg Young: "CQRS and Event Sourcing," at *Code on the Beach*, August 2014.

43. Martin Fowler: "Event Sourcing," *martinfowler.com*, December 12, 2005.

44. Vaughn Vernon: *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013. ISBN: 978-0-321-83457-7

45. H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz: "View Maintenance Issues for the Chronicle Data Model," at *14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (PODS), May 1995. doi:10.1145/212433.220201

46. "Event Store 3.5.0 Documentation," Event Store LLP, *docs.geteventstore.com*, February 2016.

47. Martin Kleppmann: *Making Sense of Stream Processing*. Report, O'Reilly Media, May 2016.

48. Sander Mak: "Event-Sourced Architectures with Akka," at *JavaOne*, September 2014.

49. Julian Hyde: personal communication, June 2016.

50. Ashish Gupta and Inderpal Singh Mumick: *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999. ISBN: 978-0-262-57122-7

51. Timothy Griffin and Leonid Libkin: "Incremental Maintenance of Views with Duplicates," at *ACM International Conference on Management of Data* (SIGMOD), May 1995. doi:10.1145/223784.223849

52. Pat Helland: "Immutability Changes Everything," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

53. Martin Kleppmann: "Accounting for Computer Scientists," *martin.kleppmann.com*, March 7, 2011.

54. Pat Helland: "Accountants Don't Use Erasers," *blogs.msdn.com*, June 14, 2007.

55. Fangjin Yang: "Dogfooding with Druid, Samza, and Kafka: Metametrics at Metamarkets," *metamarkets.com*, June 3, 2015.

56. Gavin Li, Jianqiu Lv, and Hang Qi: "Pistachio: Co-Locate the Data and Compute for Fastest Cloud Compute," *yahoohadoop.tumblr.com*, April 13, 2015.

57. Kartik Paramasivam: "Stream Processing Hard Problems – Part 1: Killing Lambda," *engineering.linkedin.com*, June 27, 2016.

58. Martin Fowler: "CQRS," *martinfowler.com*, July 14, 2011.

59. Greg Young: "CQRS Documents," *cqrs.files.wordpress.com*, November 2010.

60. Baron Schwartz: “Immutability, MVCC, and Garbage Collection,” *xaprb.com*, December 28, 2013.
61. Daniel Eloff, Slava Akhmechet, Jay Kreps, et al.: "Re: Turning the Database Inside-out with Apache Samza," Hacker News discussion, *news.ycombinator.com*, March 4, 2015.
62. “Datomic Development Resources: Excision,” Cognitect, Inc., *docs.datomic.com*.
63. “Fossil Documentation: Deleting Content from Fossil,” *fossil-scm.org*, 2016.
64. Jay Kreps: “The irony of distributed systems is that data loss is really easy but deleting data is surprisingly hard,” *twitter.com*, March 30, 2015.
65. David C. Luckham: “What’s the Difference Between ESP and CEP?,” *complexevents.com*, August 1, 2006.
66. Srinath Perera: “How Is Stream Processing and Complex Event Processing (CEP) Different?,” *quora.com*, December 3, 2015.
67. Arvind Arasu, Shivnath Babu, and Jennifer Widom: “The CQL Continuous Query Language: Semantic Foundations and Query Execution,” *The VLDB Journal*, volume 15, number 2, pages 121–142, June 2006. doi:10.1007/s00778-004-0147-z
68. Julian Hyde: “Data in Flight: How Streaming SQL Technology Can Help Solve the Web 2.0 Data Crunch,” *ACM Queue*, volume 7, number 11, December 2009. doi:10.1145/1661785.1667562
69. “Esper Reference, Version 5.4.0,” EsperTech, Inc., *espertech.com*, April 2016.
70. Zubair Nabi, Eric Bouillet, Andrew Bainbridge, and Chris Thomas: “Of Streams and Storms,” IBM technical report, *developer.ibm.com*, April 2014.
71. Milinda Pathirage, Julian Hyde, Yi Pan, and Beth Plale: “SamzaSQL: Scalable Fast Data Management with Streaming SQL,” at *IEEE International Workshop on High-Performance Big Data Computing* (HPBDC), May 2016. doi:10.1109/IPDPSW.2016.141
72. Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier: “HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm,” at *Conference on Analysis of Algorithms* (AofA), June 2007.
73. Jay Kreps: “Questioning the Lambda Architecture,” *oreilly.com*, July 2, 2014.
74. Ian Hellström: “An Overview of Apache Streaming Technologies,” *databaseline.wordpress.com*, March 12, 2016.
75. Jay Kreps: “Why Local State Is a Fundamental Primitive in Stream Processing,” *oreilly.com*, July 31, 2014.

76. Shay Banon: "Percolator," *elastic.co*, February 8, 2011.

77. Alan Woodward and Martin Kleppmann: "Real-Time Full-Text Search with Luwak and Samza," *martin.kleppmann.com*, April 13, 2015.

78. "Apache Storm 1.0.1 Documentation," *storm.apache.org*, May 2016.

79. Tyler Akidau: "The World Beyond Batch: Streaming 102," *oreilly.com*, January 20, 2016.

80. Stephan Ewen: "Streaming Analytics with Apache Flink," at *Kafka Summit*, April 2016.

81. Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, et al.: "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," at *39th International Conference on Very Large Data Bases* (VLDB), August 2013.

82. Alex Dean: "Improving Snowplow's Understanding of Time," *snowplowanalytics.com*, September 15, 2015.

83. "Windowing (Azure Stream Analytics)," Microsoft Azure Reference, *msdn.microsoft.com*, April 2016.

84. "State Management," Apache Samza 0.10 Documentation, *samza.apache.org*, December 2015.

85. Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, et al.: "Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams," at *ACM International Conference on Management of Data* (SIGMOD), June 2013. doi:10.1145/2463676.2465272

86. Martin Kleppmann: "Samza Newsfeed Demo," *github.com*, September 2014.

87. Ben Kirwin: "Doing the Impossible: Exactly-Once Messaging Patterns in Kafka," *ben.kirw.in*, November 28, 2014.

88. Pat Helland: "Data on the Outside Versus Data on the Inside," at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

89. Ralph Kimball and Margy Ross: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, 2013. ISBN: 978-1-118-53080-1

90. Viktor Klang: "I'm coining the phrase 'effectively-once' for message processing with at-least-once + idempotent operations," *twitter.com*, October 20, 2016.

91. Matei Zaharia, Tathagata Das, Haoyuan Li, et al.: "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters," at *4th USENIX Conference in Hot Topics in Cloud Computing* (HotCloud), June 2012.

92. Kostas Tzoumas, Stephan Ewen, and Robert Metzger: "High-Throughput, Low-Latency, and Exactly-Once Stream Processing with Apache Flink," *data-artisans.com*, August 5, 2015.

93. Paris Carbone, Gyula Fóra, Stephan Ewen, et al.: "Lightweight Asynchronous Snapshots for Distributed Dataflows," arXiv:1506.08603 [cs.DC], June 29, 2015.

94. Ryan Betts and John Hugg: *Fast Data: Smart and at Scale*. Report, O'Reilly Media, October 2015.

95. Flavio Junqueira: "Making Sense of Exactly-Once Semantics," at *Strata+Hadoop World London*, June 2016.

96. Jason Gustafson, Flavio Junqueira, Apurva Mehta, Sriram Subramanian, and Guozhang Wang: "KIP-98 – Exactly Once Delivery and Transactional Messaging," *cwiki.apache.org*, November 2016.

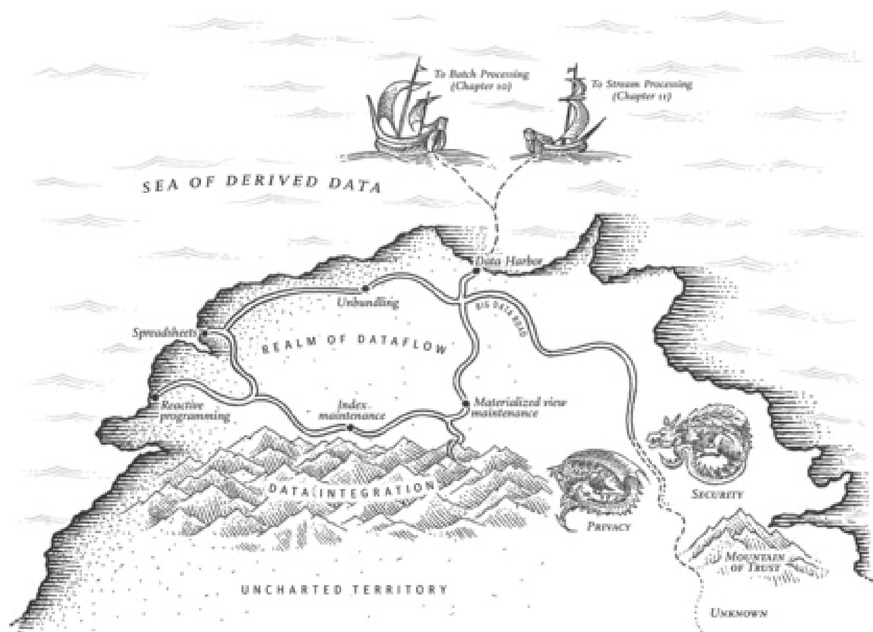
97. Pat Helland: "Idempotence Is Not a Medical Condition," *Communications of the ACM*, volume 55, number 5, page 56, May 2012. doi:10.1145/2160718.2160734

98. Jay Kreps: "Re: Trying to Achieve Deterministic Behavior on Recovery/Rewind," email to *samza-dev* mailing list, September 9, 2014.

99. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson: "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, volume 34, number 3, pages 375–408, September 2002. doi:10.1145/568522.568525

00. Adam Warski: "Kafka Streams – How Does It Fit the Stream Processing Landscape?," *softwaremill.com*, June 1, 2016.

12. 数据系统的未来



> 如果船长的终极目标是保护船只，他应该永远待在港口。
>
> ——圣托马斯·阿奎那《神学大全》（1265-1274）

[TOC]

到目前为止，本书主要描述的是目前的情况。在这最后一章中，我们将转向未来，讨论应该如何做：我将提出一些想法和方法，我相信这些方法可以从根本上改进我们设计和构建应用程序的方式。

对未来的看法和猜测当然是主观的，所以我在写这篇个人意见的时候会用到第一人称。我们欢迎您不同意并形成自己的观点，但我希望本章的观点至少可以成为一个富有成效的讨论的出发点，并为经常混淆的概念提供一些清晰。

第1章概述了本书的目标：探索如何创建可靠，可伸缩和可维护的应用程序和系统。这些主题贯穿了所有的章节：例如，我们讨论了许多有助于提高可靠性的容错算法，提高可扩展性的分区，以及提高可维护性的进化和抽象机制。在本章中，我们将把所有这些想法结合在一起，并以这些想法为基础来设想未来。我们的目标是发现如何设计比今天更好的应用程序——强大，正确，可演化，并最终对人类有益。

数据集成

本书中反复出现的主题是，对于任何给定的问题，都有几种解决方案，所有这些解决方案都有不同的优点，缺点和折衷。例如，在第3章讨论存储引擎时，我们看到了日志结构存储，B树和列存储。在第5章讨论复制时，我们看到了单领导，多领导和无领导的方法。

如果你有一个问题，例如“我想存储一些数据并稍后再查询”，那么没有一个正确的解决方案，但是在不同的情况下，每种方法都是适当的。软件实现通常必须选择一种特定的方法。要使一个代码路径健壮并且很好地尝试在一个软件中执行所有操作几乎可以保证实现效果很差，这很难。

因此，软件工具的最合适的选择也取决于情况。每一个软件，甚至是所谓的“通用”数据库，都是针对特定的使用模式而设计的。

面对如此众多的替代品，第一个挑战就是弄清楚软件产品与其所处的环境之间的映射关系。供应商可以理解地不愿意告诉你他们的软件不适合的工作负载类型，但是希望以前的章节能够给你提供一些问题，以便在各行之间阅读并更好地理解这些权衡。

但是，即使您完全理解工具与其使用环境之间的映射，还有一个挑战：在复杂的应用程序中，数据通常以多种不同的方式使用。不太可能存在适用于所有不同数据使用环境的软件，因此您不可避免地需要拼凑几个不同的软件以提供您的应用程序的功能。

组合使用衍生数据的工具

例如，为了处理任意关键字的查询，需要将OLTP数据库与全文搜索索引集成在一起是很常见的。尽管一些数据库（如PostgreSQL）包含了全文索引功能，可以满足简单的应用需求【1】，但更复杂的搜索工具需要专业的信息检索工具。相反，搜索索引通常不适合作为一个持久的记录系统，因此许多应用程序需要结合两种不同的工具来满足所有要求。

我们谈到了将数据系统集成到“使系统保持同步”（第452页）的问题。随着数据的不同表示数量的增加，集成问题变得更加困难。除了数据库和搜索索引之外，也许您需要保留分析系统（数据仓库或批处理和流处理系统）中的数据副本。维护从原始数据衍生的对象的高速缓存或非规范化版本;通过机器学习，分类，排名或推荐系统传递数据;或根据对数据的更改发送通知。

令人惊讶的是，我经常看到软件工程师做出如下陈述：“根据我的经验，99%的人只需要X”或“......不需要X”（对于X的各种值）。我认为这样的陈述更多地讲述了讲话者的经验，而不是技术的实际有用性。您可能想要对数据执行的各种操作范围非常广泛。一个人认为是一个模糊和毫无意义的功能，可能是别人的核心要求。如果缩小数据流并考虑整个组织的数据流，那么对数据集成的需求往往就会变得明显。

推理数据流

当需要在多个存储系统中维护相同数据的副本以满足不同的访问模式时，您需要非常清楚输入和输出：哪些数据先写入，哪些表示来自哪些源？如何以正确的格式将数据导入所有正确的地方？

例如，您可能会首先将数据写入记录数据库系统，捕获对该数据库所做的更改（请参阅第454页上的“捕获变更数据”），然后将更改应用于数据库中的搜索索引相同的顺序。如果更改数据捕获（CDC）是更新索引的唯一方式，则可以确定该索引完全来自记录系统，因此与其保持一致（禁止软件中的错误）。写入数据库是向该系统提供新输入的唯一方式。

允许应用程序直接写入搜索索引和数据库引入了图11-4所示的问题，其中两个客户端同时发送冲突写入，并且两个存储系统按不同顺序处理它们。在这种情况下，既不是数据库也不是

基于事件日志更新衍生数据系统通常可以做出决定性的和幂等的（参见第478页的“幂等性”），使得从故障中恢复相当容易。

衍生数据与分布式事务

保持不同数据系统彼此一致的经典方法涉及分布式事务，如第354页的“原子提交和两阶段提交（2PC）”中所述。与分布式事务相比，使用衍生数据系统的方法如何？

在抽象层面，他们通过不同的方式达到类似的目标。分布式事务通过使用锁定进行互斥来决定写入的顺序（请参阅第257页上的“两阶段锁定（2PL）”），而CDC和事件源使用日志进行排序。分布式事务使用原子提交来确保更改只生效一次，而基于日志的系统通常基于确定性重试和幂等性。

最大的不同之处在于事务系统通常提供线性（请参阅第324页的“线性一致性”），这意味着有用的保证，例如读取自己的写入（请参阅第162页的“读已之写”）。另一方面，衍生数据系统通常是异步更新的，因此它们不会默认提供相同的时间保证。

在愿意支付分布式交易成本的有限环境中，它们已被成功使用。但是，我认为XA的容错能力和性能特征较差（请参阅第364页的“实践中的分布式事务”），这严重限制了它的实用性。我相信有可能为分布式事务创建一个更好的协议，但是将这样一个协议广泛采用并与现有工具集成将是具有挑战性的，并且不太可能很快发生。

在没有广泛支持良好分布式事务协议的情况下，我认为基于日志的衍生数据是集成不同数据系统的最有前途的方法。然而，诸如阅读自己写作的保证是有用的，我认为告诉每个人“最终的一致性是不可避免的 —— 吸收它并学会处理它”是没有成果的（至少不是没有良好的指导如何处理它）。

在第51页的“瞄准正确性”中，我们将讨论一些在异步衍生系统之上实现更强保障的方法，并在分布式事务和异步基于对数系统之间建立中间立场。

全局有序的限制

对于足够小的系统，构建一个完全有序的事件日志是完全可行的（正如单主复制数据库的流行所证明的那样，这正好构建了这样一个日志）。但是，随着系统向更大更复杂的工作负载扩展，限制开始出现：

- 在大多数情况下，构建完全有序的日志需要所有事件通过决定订购的单个领导节点。如果事件吞吐量大于单台计算机可处理的事件，则需要将其分割到多台计算机上（请参见第446页的“分区日志”）。然后，两个不同分区中的事件顺序不明确。
- 如果服务器分布在多个地理位置分散的数据中心上，例如为了容忍整个数据中心脱机，您通常在每个数据中心都有单独的领导者，因为网络延迟会导致同步的跨数据中心协调效率低下（请参阅“多主复制“）。这意味着源自两个不同数据中心的事件的未定义排序。
- 将应用程序部署为微服务时（请参阅第125页上的“通过服务进行数据流：REST和RPC”），常见的设计选择是将每个服务及其持久状态作为独立单元进行部署，服务之间不共享持久状态。当两个事件来自不同的服务时，这些事件没有定义的顺序。
- 某些应用程序保持客户端状态，该状态在用户输入时立即更新（无需等待服务器确认），甚至可以继续脱机工作（请参阅第170页的“脱机操作的客户端”）。有了这样的应用程序，客户端和服务器很可能以不同的顺序看到事件。

在形式上，决定事件的总次序称为总次序广播，相当于共识（请参阅第366页上的“共识算法和全序广播”）。大多数共识算法都是针对单个节点的吞吐量足以处理整个事件流的情况而设计的，并且这些算法不提供多个节点共享事件排序工作的机制。设计共识算法仍然是一个开放的研究问题，它可以扩展到单个节点的吞吐量之外，并且在地理上分散的环境中工作良好。

订购事件以捕捉因果关系

在事件之间不存在因果关系的情况下，缺乏全部命令并不是一个大问题，因为并发事件可以任意排序。其他一些情况很容易处理：例如，当同一对象有多个更新时，它们可以通过将特定对象ID的所有更新路由到相同的日志分区来完全排序。然而，因果关系有时会以更微妙的方式出现（另请参阅“顺序和因果关系”第319页）。

例如，考虑一个社交网络服务，以及两个相互关系但刚分手的用户。其中一个用户将另一个作为朋友移除，然后向其余的朋友发送消息，抱怨他们的前伴侣。用户的意图是他们的前配偶不应该看到粗鲁的信息，因为信息是在朋友状态被撤销后发送的。

但是，在一个地方存储友谊状态并在另一个地方存储消息的系统中，不友好事件和消息发送事件之间的顺序依赖关系可能会丢失。如果未捕获到因果依赖关系，则发送有关新消息的通知的服务可能会在不友好事件之前处理消息发送事件，从而错误地向前伙伴发送通知。

在本例中，通知实际上是消息和好友列表之间的连接，使得它与我们先前讨论的连接的时间问题有关（请参阅第475页的“连接的时间依赖性”）。不幸的是，这个问题似乎并没有一个简单的答案【2,3】。起点包括：

- 逻辑时间戳可以提供没有协调的全部订购（请参见“序列号排序”第页343），因此它们可能有助于全序广播不可行的情况。但是，他们仍然要求收件人处理不按顺序发送的事件，并且需要传递其他元数据。
- 如果您可以记录一个事件来记录用户在做出决定之前所看到的系统状态，并给该事件一个唯一的标识符，那么后面的任何事件都可以引用该事件标识符来记录因果关系【4】。我们将在第513页的“读也是事件”中回到这个想法。
- 冲突解决算法（请参阅“自动冲突解决”（第165页））有助于处理以意外顺序传递的事件。它们对于维护状态很有用，但如果行为有外部副作用（例如，也许，随着时间的推移，应用程序开发模式将出现，使得能够有效地捕获因果依赖关系，并且保持正确的衍生状态，而不会迫使所有事件经历全部命令广播的瓶颈。

批量处理与流处理

我会说数据集成的目标是确保数据在所有正确的地方以正确的形式结束。这样做需要消耗投入，转化，加入，过滤，汇总，培训模型，评估并最终写入适当的输出。批处理和流处理器是实现这一目标的工具。

批处理和流处理的输出是衍生的数据集，例如搜索索引，实例化视图，向用户显示的建议，聚合度量等（请参阅“批处理工作流的输出”（第417页）和“流处理的用法”第465页）。

正如我们在第10章和第11章中看到的，批处理和流处理有许多共同的原则，主要的根本区别在于流处理器在无界数据集上运行，而批处理输入是已知的有限大小。处理引擎的实现方式也有很多细节上的差异，但是这些区别开始模糊。

Spark在批处理引擎上执行流处理，将流分解为微格式，而Apache Flink则在流处理引擎上执行批处理【5】。原则上，一种类型的处理可以在另一种类型上仿真，但是性能特征会有所不同：例如，在跳跃或滑动窗口时，微博可能表现不佳【6】。

保持衍生状态

批处理具有非常强大的功能特性（即使代码不是用函数式编程语言编写的）：它鼓励确定性的纯函数，其输出仅依赖于输入，除了显式输出外没有副作用，处理输入作为不可变的，并作为附加的输出。流处理类似，但它扩展了运算符以允许受管理的容错状态（请参阅第478页的“重建失败后的状态”）。

具有良好定义的输入和输出的确定性函数的原理不仅有利于容错（请参见第478页的“幂等性”），但也简化了有关组织中数据流的推理【7】。无论衍生数据是搜索索引，统计模型还是缓存，从数据管道角度来看，从另一个衍生出一件事情，通过功能应用程序代码推送一个系统中的状态更改和应用对衍生系统的影响。

原则上，衍生数据系统可以同步维护，就像关系数据库在与被索引表写入操作相同的事务中同步更新辅助索引一样。然而，异步是基于事件日志的系统稳健的原因：它允许系统的一部分故障被本地包含，而如果任何一个参与者失败，分布式事务将中止，因此他们倾向于通过

将故障扩展到系统的其余部分（请参阅第363页的“分布式事务的限制”）。

我们在第206页的“分区和二级索引”中看到，二级索引经常跨越分区边界。具有二级索引的分区系统需要将写入发送到多个分区（如果索引是分词）或将读取发送到所有分区（如果索引是文档分区的话）。如果索引是异步维护的，这种交叉分区通信也是最可靠和可扩展的【8】（另请参阅“多分区数据处理”第479页）。

为应用程序演变重新处理数据

在维护衍生数据时，批处理和流处理都是有用的。流处理允许将输入中的变化以低延迟反映在衍生视图中，而批处理允许重新处理大量累积的历史数据以便将新视图导出到现有数据集上。

特别是，重新处理现有数据为维护系统提供了一个良好的机制，并将其发展为支持新功能和变更需求（参见第4章）。如果不进行重新处理，模式演化就会局限于简单的变化，例如向记录中添加新的可选字段或添加新类型的记录。无论是在写模式还是在读模式中都是如此（请参阅第39页的“文档模型中的模式灵活性”）。另一方面，通过重新处理，可以将数据集重组为一个完全不同的模型，以便更好地满足新的要求。

> ### 在铁路上的模式迁移
>
> 大规模的“模式迁移”也发生在非计算机系统中。例如，在19世纪英国铁路建设初期，轨距（两轨之间的距离）就有了各种各样的交易标准。为一个测量仪而建的列车不能在另一个测量仪的轨道上运行，这限制了火车网络中可能的相互连接【9】。
>
> 在1846年最终确定了一个标准仪表之后，其他仪表的轨道必须转换 —— 但是如何在不关闭火车线路的情况下进行数月甚至数年？解决的办法是首先将轨道转换为双轨或混合轨距，方法是增加第三轨。这种转换可以逐渐完成，当完成时，两个仪表的列车可以在三条轨道中的两条轨道上运行。事实上，一旦所有的列车都转换成标准轨距，那么可以移除提供非标准轨距的轨道。
>
> 以这种方式“再加工”现有的轨道，让新旧版本并存，可以在几年的时间内逐渐改变轨距。然而，这是一项昂贵的事业，这就是今天非标准仪表仍然存在的原因。例如，旧金山湾区的BART系统使用与美国大部分地区不同的仪表。

衍生视图允许逐步演变。如果您想重新构建数据集，则不需要执行迁移作为突然切换。相反，您可以将旧架构和新架构并排维护为相同基础数据上的两个独立衍生视图。然后，您可以开始将少量用户转移到新视图，以测试其性能并发现任何错误，而大多数用户仍然会被路由到旧视图。逐渐地，您可以增加访问新视图的用户比例，最终您可以删除旧视图【10】。

这种逐渐迁移的美妙之处在于，如果出现问题，每个阶段的过程都很容易逆转：您始终有一个可以回到的工作系统。通过降低不可逆损害的风险，您可以更有信心继续前进，从而更快地改善您的系统【11】。

Lambda架构

如果批处理用于重新处理历史数据，并且流处理用于处理最近的更新，那么您如何将这两者结合起来？拉姆达体系结构【12】是这方面的一个建议，引起了很多关注。

lambda体系结构的核心思想是通过将不可变事件附加到不断增长的数据集来记录传入数据，这类似于事件源（请参阅第457页上的“事件源”）。从这些事件中，推导出读取优化的视图。Lambda体系结构建议并行运行两个不同的系统：批处理系统（如Hadoop MapReduce）和独立的流处理系统（如Storm）。

在lambda方法中，流处理器消耗事件并快速生成对视图的近似更新;批处理器稍后将使用同一组事件并生成衍生视图的更正版本。这个设计背后的原因是批处理更简单，因此不易出错，而流处理器被认为是不太可靠和难以容错的（请参阅“故障容错”）。而且，流处理可以使用快速近似算法，而批处理使用较慢的精确算法。

拉姆达体系结构是一个有影响力的想法，它将数据系统的设计变得更好，尤其是通过推广将视图衍生到不可变事件流和在需要时重新处理事件的原则。但是，我也认为它有一些实际问题：

- 必须保持相同的逻辑才能在批处理和流处理框架中运行，这是额外的工作。虽然像Summingbird 【13】这样的库提供了一个抽象的计算，可以在一个批处理或流的上下文中运行，调试，调整和维护两个不同系统的操作复杂性仍然【14】。
- 由于流管道和批处理管道产生单独的输出，因此需要合并它们以响应用户请求。如果计算是通过滚动窗口的简单聚合，则合并相当容易，但如果使用更复杂的操作（例如连接和会话化）导出视图，或者输出不是时间序列，则显得非常困难。
- 尽管有能力重新处理整个历史数据集是很好的，但在大型数据集上这样做经常会很昂贵。因此，批处理流水线通常需要设置为处理增量批处理（例如，在每小时结束时处理一小时的数据），而不是重新处理所有内容。这引发了第468页的“关于时间的推理”中讨论的问题，例如处理分段器和处理跨批次边界的窗口。增加批量计算会增加复杂性，使其更类似于流式传输层，这与保持批处理层尽可能简单的目标背道而驰。

统一批处理和流处理

最近的工作使得lambda体系结构的优点在没有其缺点的情况下得以实现，允许批处理计算（重新处理历史数据）和流计算（处理事件到达时）在同一个系统中实现【15】。

在一个系统中统一批处理和流处理需要以下功能，这些功能越来越广泛：

- 通过处理最近事件流的相同处理引擎来重放历史事件的能力。例如，基于日志的消息代理可以重放消息（请参阅第451页的“重放旧消息”），某些流处理器可以从HDFS等分布式文件系统读取输入。
- 对于流处理器来说，只有一次语义 —— 即确保输出与未发生故障的输出相同，即使事实上发生故障（请参阅“故障容错”（第476页））。与批处理一样，这需要丢弃任何失败任务的部分输出。

- 按事件时间进行窗口化的工具，而不是按处理时间进行窗口化，因为处理历史事件时的处理时间毫无意义（请参阅第468页的“关于时间的推理”）。例如，Apache Beam提供了用于表达这种计算的API，然后可以使用Apache Flink或Google Cloud Dataflow运行。

拆分数据库

在最抽象的层面上，数据库，Hadoop和操作系统都执行相同的功能：它们存储一些数据，并允许您处理和查询数据【16】。数据库将数据存储在某些数据模型（表中的文档，文档中的顶点，图形中的顶点等）的记录中，而操作系统的文件系统则将数据存储在文件中——但在其核心上，都是“信息管理”系统【17】。正如我们在第10章中看到的，Hadoop生态系统有点像Unix的分布式版本。

当然，有很多实际的差异。例如，许多文件系统不能很好地处理包含1000万个小文件的目录，而包含1000万个小记录的数据库是完全正常且不起眼的。无论如何，操作系统和数据库之间的相似之处和差异值得探讨。

Unix和关系数据库以非常不同的哲学来处理信息管理问题。 Unix认为它的目的是为程序员提供一种逻辑但相当低层次的硬件抽象，而关系数据库则希望为应用程序员提供一种高层次的抽象，以隐藏磁盘上数据结构的复杂性，并发性，崩溃恢复以及等等。 Unix开发的管道和文件只是字节序列，而数据库则开发了SQL和事务。

哪种方法更好？当然，这取决于你想要的。 Unix是“简单的”，因为它是硬件资源相当薄的包装;关系数据库是“更简单”的，因为一个简短的声明性查询可以利用很多强大的基础设施（查询优化，索引，连接方法，并发控制，复制等），而不需要查询作者理解实施细节。

这些哲学之间的矛盾已经持续了几十年（Unix和关系模型都出现在70年代初），仍然没有解决。例如，我将NoSQL运动解释为希望将低级别抽象方法应用于分布式OLTP数据存储领域。

在这一部分我将试图调和这两个哲学，希望我们可以结合两全其美。

组合使用数据存储技术

在本书的过程中，我们讨论了数据库提供的各种功能及其工作原理，其中包括：

- 二级索引，使您可以根据字段的值有效地搜索记录（请参阅第79页上的“其他索引结构”）
- 物化视图，这是一种预先计算的查询结果缓存（请参阅“聚合：数据立方体和物化视图”，第101页）
- 复制日志，保持其他节点上数据的副本最新（请参阅第158页中的“复制日志的实现”）
- 全文搜索索引，允许在文本中进行关键字搜索（请参见第88页上的“全文搜索和模糊索引”）以及内置于某些关系数据库【1】

在第十章和第十一章中，出现了类似的主题。我们讨论了如何构建全文搜索索引（请参阅第357页上的“批处理工作流的输出”），了解有关实例化视图维护（请参阅“维护实例化视图”一节第437页）以及有关将更改从数据库复制到衍生数据系统（请参阅第454页的“更改数据捕获”）。

数据库中内置的功能与人们用批处理和流处理器构建的衍生数据系统似乎有相似之处。

创建一个索引

想想当你运行 `CREATE INDEX` 在关系数据库中创建一个新的索引时会发生什么。数据库必须扫描表的一致性快照，挑选出所有被索引的字段值，对它们进行排序，然后写出索引。然后它必须处理自一致快照以来所做的写入操作（假设表在创建索引时未被锁定，所以写操作可能会继续）。一旦完成，只要事务写入表中，数据库就必须继续保持索引最新。

此过程非常类似于设置新的追随者副本（请参阅第155页的“设置新的追随者”），也非常类似于流系统中的引导更改数据捕获（请参阅第455页的“初始快照”）。

无论何时运行CREATE INDEX，数据库都会重新处理现有数据集（如第494页的“重新处理应用程序数据的演变数据”中所述），并将该索引作为新视图导出到现有数据上。现有数据可能是状态的快照，而不是所有发生变化的日志，但两者密切相关（请参阅“状态，数据流和不变性”第459页）。

一切的元数据库

有鉴于此，我认为整个组织的数据流开始像一个巨大的数据库【7】。每当批处理，流或ETL过程将数据从一个地方传输到另一个地方并形成表单时，就像数据库子系统一样，使索引或物化视图保持最新。

像这样看，批处理和流处理器就像触发器，存储过程和物化视图维护例程的精细实现。他们维护的衍生数据系统就像不同的索引类型。例如，关系数据库可能支持B树索引，散列索引，空间索引（请参阅第79页的“多列索引”）以及其他类型的索引。在新兴的衍生数据系统体系结构中，不是将这些设施作为单个集成数据库产品的功能实现，而是由各种不同的软件提供，运行在不同的机器上，由不同的团队管理。

这些发展在未来将会把我们带到哪里？如果我们从没有适合所有访问模式的单一数据模型或存储格式的前提出发，我推测有两种途径可以将不同的存储和处理工具组合成一个有凝聚力的系统：

联合数据库：统一读取

可以为各种各样的底层存储引擎和处理方法提供一个统一的查询接口 —— 一种称为联邦数据库或多存储的方法【18,19】。例如，PostgreSQL的外部数据包装功能符合这种模式【20】。需要专用数据模型或查询接口的应用程序仍然可以直接访问底层存储引擎，而想要组合来自不同位置的数据的用户可以通过联合接口轻松完成操作。

联合查询接口遵循单一集成系统的关系传统，具有高级查询语言和优雅的语义，但却是一个复杂的实现。

非捆绑数据库：统一写入

虽然联合会解决了跨多个不同系统的只读查询问题，但它并没有很好的解决跨系统同步写入的问题。我们说过，在单个数据库中，创建一致的索引是一项内置功能。当我们构建多个存储系统时，我们同样需要确保所有数据更改都会在所有正确的位置结束，即使在出现故障时也是如此。将存储系统可靠地插接在一起（例如，通过更改数据捕获和事件日志）更容易，就像将数据库的索引维护功能以可以跨不同技术同步写入的方式分开【7,21】。

非捆绑方法遵循Unix传统的小型工具，它可以很好地完成一件事【22】，通过统一的低级API（管道）进行通信，并且可以使用更高级别的语言（shell）【16】 。

开展分拆工作

联邦和非捆绑是同一个硬币的两个方面：用不同的组件构成可靠，可扩展和可维护的系统。联合只读联邦和非捆绑是同一个硬币的两个方面：用不同的组件构成可靠，可扩展和可维护的系统。联合只读查询需要将一个数据模型映射到另一个数据模型，这需要一些思考，但最终还是一个可管理的问题。我认为保持写入到几个存储系统是同步的更困难的工程问题，所以我将重点关注它。

传统的同步写入方法需要跨异构存储系统的分布式事务【18】，我认为这是错误的解决方案（请参阅“导出的数据与分布式事务”第495页）。单个存储或流处理系统内的事务是可行的，但是当数据跨越不同技术之间的边界时，我认为具有幂等写入的异步事件日志是一种更加健壮和实用的方法。

例如，分布式事务在某些流处理器中使用，以精确匹配一次语义（请参阅第477页的“重新访问原子提交”），这可以很好地工作。然而，当事务需要涉及由不同人群编写的系统时（例如，当数据从流处理器写入分布式键值存储或搜索索引时），缺乏标准化的事务协议会使集成更难。有幂等消费者的事件的有序日志（参见第478页的“幂等性”）是一种更简单的抽象，因此在异构系统中实现更加可行【7】。

基于日志的集成的一大优势是各个组件之间的松散耦合，这体现在两个方面：

1. 在系统级别，异步事件流使整个系统对各个组件的中断或性能下降更加稳健。如果使用者运行缓慢或失败，那么事件日志可以缓冲消息（请参阅“磁盘空间使用情况”第369页），以便生产者和任何其他使用者可以继续不受影响地运行。有问题的消费者可以在固定时赶上，因此不会错过任何数据，并且包含故障。相比之下，分布式事务的同步交互往往会将本地故障升级为大规模故障（请参见第363页的“分布式事务的限制”）。
2. 在人力方面，分拆数据系统允许不同的团队独立开发，改进和维护不同的软件组件和服务。专业化使得每个团队都可以专注于做好一件事，并与其他团队的系统进行明确的界面。事件日志提供了一个足够强大的接口，以捕获相当强的一致性属性（由于持久性和事件的顺序），但也足够普遍适用于几乎任何类型的数据。

非捆绑与集成系统

如果分拆确实成为未来的方式，它将不会取代目前形式的数据库 —— 它们仍然会像以往一样需要。数据库仍然需要维护流处理器中的状态，并且为批处理和流处理器的输出提供查询服务（请参阅第419页上的“批处理工作流的输出”和第464页上的“处理流”）。专门的查询引擎将继续对特定的工作负载非常重要：例如，MPP数据仓库中的查询引擎针对探索性分析查询进行了优化，并且能够很好地处理这种类型的工作负载（请参阅第417页的“将Hadoop与分布式数据库进行比较”。

运行几个不同基础架构的复杂性可能是一个问题：每一个软件都有一个学习曲线，配置问题和操作怪癖，因此值得部署尽可能少的移动部件。与由应用程序代码【23】组成的多个工具组成的系统相比，单一集成软件产品也可以在其设计的工作负载类型上实现更好，更可预测的性能。正如我在前言中所说的那样，为了扩大规模而建设你不需要的是浪费精力，并且可能会将你锁定在一个不灵活的设计中。实际上，这是一种过早优化的形式。

分拆的目标不是要针对个别数据库与特定工作负载的性能进行竞争;我们的目标是允许您结合多个不同的数据库，以便在比单个软件可能实现的更广泛的工作负载范围内实现更好的性能。这是关于广度，而不是深度 —— 与我们在第414页上的“比较Hadoop与分布式数据库”中讨论的存储和处理模型的多样性一样。

因此，如果有一项技术可以满足您的所有需求，那么您最好使用该产品，而不是试图用低级组件重新实现它。只有当没有单一软件满足您的所有需求时，才会出现拆分和合成的优势。

少了什么东西？

用于组成数据系统的工具正在变得越来越好，但我认为缺少一个主要部分：我们还没有Unix shell的非捆绑式数据库（即，用于组成存储和处理系统的高级语言简单和陈述的方式）。

例如，如果我们可以简单地声明 `mysql |elasticsearch` ，类似于Unix管道【22】，这将成为 `CREATE INDEX` 的非捆绑等价物：它将MySQL数据库中的所有文档并将其索引到Elasticsearch集群中。然后它会不断捕获对数据库所做的所有更改，并自动将它们应用于搜索索引，而无需编写自定义应用程序代码。几乎任何类型的存储或索引系统都可以实现这种集成。

同样，能够更容易地预先计算和更新缓存将是一件好事。回想一下，物化视图本质上是一个预先计算的缓存，所以您可以通过为复杂查询声明指定物化视图来创建缓存，包括图上的递归查询（请参阅第49页上的“图数据模型”）和应用逻辑。在这方面有一些有趣的早期研究，如差异数据流【24,25】，我希望这些想法能够在生产系统中找到自己的方法。

围绕数据流设计应用

通过使用应用程序代码组成专门的存储和处理系统来分离数据库的方法也被称为“数据库内外”方法【26】，在2014年的一次会议演讲标题之后【27】。然而，称它为“新架构”太宏大。我把它看作是一个设计模式，一个讨论的起点，我们只是简单地给它起一个名字，以便我们

可以更好地谈论它。

这些想法不是我的;他们只是我认为我们应该学习的其他人思想的融合。尤其是，数据流语言（如Oz【28】和Juttle【29】），功能反应式编程（FRP）语言（如Elm【30,31】）和逻辑编程语言（如Bloom [32。 Jay Kreps 【7】提出了在这种背景下解绑的术语。

即使电子表格也具有比大多数主流编程语言遥远的数据流编程功能【33】。在电子表格中，可以将公式放入一个单元格中（例如，另一列中的单元格总和），并且只要公式的任何输入发生更改，公式的结果都会自动重新计算。这正是我们在数据系统级所需要的：当数据库中的记录发生更改时，我们希望自动更新该记录的任何索引，并且自动刷新依赖于记录的任何缓存视图或聚合。您不必担心这种刷新如何发生的技术细节，但能够简单地相信它可以正常工作。

因此，我认为绝大多数数据系统仍然可以从VisiCalc在1979年已经具备的功能中学习【34】。与电子表格的不同之处在于，今天的数据系统需要具有容错性，可扩展性以及持久存储数据。他们还需要能够整合不同人群编写的不同技术，并重用现有的库和服务：期望使用某种特定语言，框架或工具开发所有软件是不切实际的。

在本节中，我将详细介绍这些想法，并探讨一些围绕非捆绑数据库和数据流的想法构建应用程序的方法。

应用程序代码作为衍生函数

当一个数据集来自另一个数据集时，它会经历某种转换函数。例如：

- 辅助索引是一种具有直接转换函数的衍生数据集：对于基表中的每一行或文档，它挑选被索引的列或字段中的值，并按这些值排序（假设B树或SSTable索引，按键排序，如第3章所述）。
- 通过应用各种自然语言处理功能（如语言检测，分词，词干或词汇化，拼写纠正和同义词识别）创建全文搜索索引，然后构建用于高效查找的数据结构（例如作为倒排索引）。
- 在机器学习系统中，我们可以将模型视为通过应用各种特征提取和统计分析功能从训练数据中导出。当模型应用于新的输入数据时，模型的输出是从输入和模型（因此间接地从训练数据）中导出的。
- 缓存通常包含将以用户界面（UI）显示的形式的数据聚合。因此填充缓存需要知道UI中引用的字段；UI中的更改可能需要更新缓存填充方式的定义以及重建缓存。

辅助索引的衍生函数通常是必需的，因此它作为核心特性被构建到许多数据库中，您可以仅通过说 `CREATE INDEX` 来调用它。对于全文索引，常见语言的基本语言特征可能内置到数据库中，但更复杂的特征通常需要特定于域的调整。在机器学习中，特征工程是众所周知的特定于应用程序的特征，并且通常必须包含关于应用程序的用户交互和部署的详细知识【35】。

当创建衍生数据集的函数不是像创建二级索引那样的标准Cookie切割函数时，需要自定义代码来处理特定于应用程序的方面。而这个自定义代码是许多数据库难以抗争虽然关系数据库通常支持触发器，存储过程和用户定义的函数，它们可以用来在数据库中执行应用程序代码，但它们在数据库设计中已经有所反应了（请参阅“传输事件流”（第447页））。

应用程序代码和状态的分离

理论上，数据库可以是任意应用程序代码的部署环境，如操作系统。但是，实际上他们已经变得不适合这个目的。它们不适合现代应用程序开发的要求，例如依赖性和软件包管理，版本控制，滚动升级，可演化性，监控，指标，对网络服务的调用以及与外部系统的集成。

另一方面，Mesos，YARN，Docker，Kubernetes等部署和集群管理工具专为运行应用程序代码而设计。通过专注于做好一件事情，他们能够做得比将数据库作为其众多功能之一执行用户定义的功能要好得多。 我认为让系统的某些部分专门用于持久数据存储以及专门运行应用程序代码的其他部分是有意义的。这两者可以在保持独立的同时互动。

现在大多数Web应用程序都是作为无状态服务部署的，其中任何用户请求都可以路由到任何应用程序服务器，并且服务器在发送响应后会忘记所有请求。这种部署方式很方便，因为可以随意添加或删除服务器，但状态必须到某个地方：通常是数据库。趋势是将无状态应用程序逻辑与状态管理（数据库）分开：不将应用程序逻辑放入数据库中，也不将持久状态置于应用程序中【36】。正如职能规划界人士喜欢开玩笑说的那样，“我们相信教会与国家的分离”【37】[i]

> i. 解释一个笑话很少改进它，但我不想让任何人感到被遗漏。 在这里，Church是数学家Alonzo Church的参考资料，他创建了lambda演算，这是早期的计算形式，是大多数函数式编程语言的基础。 lambda演算不具有可变状态（即没有变量可以被覆盖），所以可以说可变状态与Church的工作是分开的。 ↩

在这个典型的Web应用程序模型中，数据库充当一种可以通过网络同步访问的可变共享变量。应用程序可以读取和更新变量，并且数据库负责保持持久性，提供一些并发控制和容错。

但是，在大多数编程语言中，您无法订阅可变变量中的更改 —— 您只能定期读取它。与电子表格不同，如果变量的值发生变化，变量的读者不会收到通知。（您可以在自己的代码中实现这样的通知 —— 这被称为观察者模式 —— 但大多数语言没有将此模式作为内置功能。）

数据库继承了这种可变数据的被动方法：如果你想知道数据库的内容是否发生了变化，通常你唯一的选择就是轮询（即定期重复你的查询）。 订阅更改只是刚刚开始出现的功能（请参阅第455页的“更改流的API支持”）。

数据流：状态变化和应用程序代码之间的相互影响

从数据流的角度思考应用意味着重新谈判应用代码和状态管理之间的关系。我们不是将数据库视为被应用程序操纵的被动变量，而是更多地考虑状态，状态更改和处理它们的代码之间的相互作用和协作。应用程序代码通过在另一个地方触发状态更改来响应状态更改。

我们在第451页的“数据库和数据流”中看到了这一思路，我们讨论将数据库更改的日志作为我们可以指定的事件流处理。消息传递系统（如角色）（请参阅第136页的“消息传递数据流”）也具有响应事件的概念。早在20世纪80年代，元组空间模型探索表示分布式计算的过程，观察状态变化并对它们做出反应【38,39】。

如前所述，当触发器由于数据更改而触发时，或者次级索引更新以反映索引表中的更改时，数据库内部会发生类似的情况。分解数据库意味着将此想法应用于在主数据库之外创建衍生数据集：缓存，全文搜索索引，机器学习或分析系统。我们可以为此使用流处理和消息传递系统。

需要记住的重要一点是，维护衍生数据与传统设计消息传递系统的异步作业执行不同（请参阅第448页上的“与传统消息传递相比的日志”）：

•在维护衍生数据时，状态更改的顺序通常很重要（如果多个视图是从事件日志衍生的，则需要按照相同的顺序处理事件，以便它们保持一致）。如第445页上的“确认和重新传递”中所述，许多消息代理在重新传送未确认消息时没有此属性。双重写入也被排除（请参阅第454页上的“保持系统同步”）。

•容错是导出数据的关键：仅丢失单个消息会导致衍生数据集永远与其数据源不同步。消息传递和衍生状态更新都必须可靠。例如，许多角色系统默认在内存中维护角色状态和消息，所以如果运行角色的机器崩溃，他们就会丢失。

稳定的消息排序和容错消息处理是相当严格的要求，但与分布式事务相比，它们更便宜，操作更稳定。现代流处理器可以提供这些排序和可靠性保证，并允许应用程序代码作为流操作符运行。此应用程序代码可以执行数据库中内置的衍生函数通常不提供的任意处理。就像管道链接的Unix工具一样，流操作符可以组成数据流周围的大型系统。每个运算符将状态变化流作为输入，并产生其他状态变化流作为输出。

流处理器和服务

当前流行的应用程序开发风格涉及将功能分解为一组通过同步网络请求（如REST API）进行通信的服务（请参阅第121页的“通过服务实现数据流：REST和RPC”）。这种面向服务的体系结构优于单个单一应用程序的优势主要在于通过松散耦合的组织可伸缩性：不同的团队可以在不同的服务上工作，从而减少团队之间的协调工作（只要服务可以独立部署和更新）。

将流操作符合并到数据流系统中与微服务方法有很多相似的特征【40】。但是，底层的通信机制是非常不同的：单向异步消息流而不是同步请求/响应交互。

除了第136页上的“消息传递数据流”中列出的优点（如更好的容错性），数据流系统还可以获得更好的性能。例如，假设客户正在购买以一种货币定价但以另一种货币支付的商品。为了执行货币转换，您需要知道当前的汇率。这个操作可以通过两种方式实现【40,41】：

1. 在微服务方法中，处理购买的代码可能会查询汇率服务或数据库以获取特定货币的当前汇率。
2. 在数据流方法中，处理采购的代码将提前订阅汇率更新流，并在当地数据库发生更改时将当前汇率记录下来。处理采购时，只需查询本地数据库即可。

第二种方法已经将同步网络请求替换为对本地数据库进行查询的另一服务（即使在同一个进程中，该请求也可能在同一台机器上）[ii]。数据流不仅方法更快，而且更稳健到另一项服务的失败。最快和最可靠的网络请求根本就没有网络请求！我们现在不使用RPC，而是在购买事件和汇率更新事件之间建立流联接（请参阅第473页的“流表联接（流增强）”）。

> [ii]. 在微服务方法中，您可以通过在处理购买的服务中本地缓存汇率来避免同步网络请求。 但是，为了使缓存保持新鲜，您需要定期轮询更新的汇率，或订阅更改流——这正是数据流方法中发生的情况。↩

加入是时间相关的：如果购买事件在稍后的时间点被重新处理，汇率将会改变。如果要重建原始输出，则需要获取原始购买时的历史汇率。无论您是查询服务还是订阅汇率更新流，您都需要处理这种时间依赖性（请参阅第475页的“连接的时间依赖性”）。

订阅一系列更改，而不是在需要时查询当前状态，使我们更接近类似电子表格的计算模型：当某些数据发生更改时，依赖于此的所有衍生数据都可以快速更新。还有很多未解决的问题，例如围绕时间依赖连接等问题，但我认为围绕数据流想法构建应用程序是一个非常有希望的方向。

观察衍生数据状态

在抽象层面，上一节讨论的数据流系统为您提供了创建衍生数据集（例如搜索索引，物化视图和预测模型）并使其保持最新的过程。让我们称这个过程为写入路径：只要某些信息被写入系统，它可能会经历批处理和流处理的多个阶段，并且最终每个衍生数据集都会更新以合并写入的数据。图12-1显示了更新搜索索引的示例。

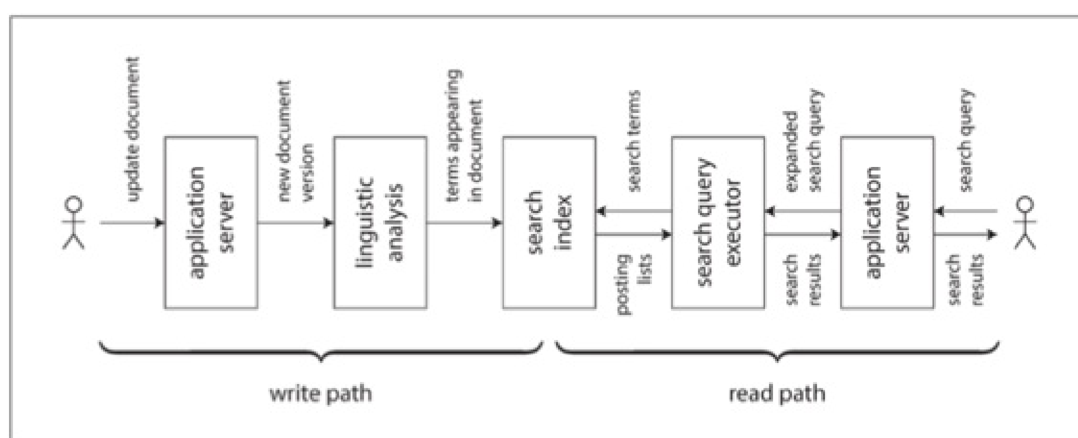


图**12-1** 在搜索索引中，写入（文档更新）符合读取（查询）

但为什么你首先创建衍生数据集？很可能是因为你想在以后再次查询它。这是读取路径：在提供从衍生数据集中读取的用户请求时，可能会对结果执行一些更多处理，然后构建对用户的响应。

总而言之，写入路径和读取路径涵盖了数据的整个旅程，从收集数据的地步到使用数据（可能是由另一个人）。写入路径是预先计算的行程的一部分 —— 即，一旦数据进入，即刻完成，无论是否有人要求查看它。阅读路径是旅程中只有当有人要求时才会发生的部分。如果您熟悉函数式编程语言，则可能会注意到写入路径类似于急切的评估，读取路径类似于懒惰评估。

如图12-1所示，衍生数据集是写入路径和读取路径相遇的地方。它代表了在写入时需要完成的工作量与在读取时需要完成的工作量之间的权衡。

物化视图和缓存

全文搜索索引就是一个很好的例子：写路径更新索引，读路径在索引中搜索关键字。读写都需要做一些工作。写入需要更新文档中出现的所有术语的索引条目。阅读需要搜索查询中的每个单词，并应用布尔逻辑来查找包含查询中所有单词（AND运算符）的文档，或者每个单词（OR运算符）的任何同义词。

如果您没有索引，搜索查询将不得不扫描所有文档（如grep），如果您有大量文档，这将会非常昂贵。没有索引意味着写入路径上的工作量较少（没有要更新的索引），但是在读取路径上需要更多工作。

> iii. 假设一个有限的语料库，那么非空洞的搜索结果的非空搜索结果是有限的。然而，它在语料库中的术语数量是指数级的，这仍然是一个坏消息。↩

另一方面，您可以想象为所有可能的查询预先计算搜索结果。在这种情况下，读取路径上的工作量会减少：不需要布尔逻辑，只需查找查询结果并返回即可。但是，写入路径会更加昂贵：可能要求的可能的搜索查询集合是无限的，因此预先计算所有可能的搜索结果将需要无限的时间和存储空间。那不会很好[iii]。

另一个选择是预先计算搜索结果，只对一组固定的最常见的查询进行计算，以便它们可以快速地服务而不必去索引。不寻常的查询仍然可以从索引提供。这通常会被称为常见查询缓存，尽管我们也可以称之为物化

视图，因为当新文档出现时，需要更新这些文档，这些文档应该包含在其中一个常见查询的结果中。

从这个例子中我们可以看到，索引不是写路径和读路径之间唯一可能的边界。常见搜索结果的缓存是可能的，并且在少量文档上也可以使用没有索引的类似grep的扫描。如此看来，缓存，索引和物化视图的作用很简单：它们改变了读取路径和写入路径之间的边界。通过预先计算结果，它们允许我们在写入路径上做更多的工作，以节省读取路径上的工作量。

在写作路径上完成的工作和读取路径之间的界限实际上是本书开始处的Twitter示例的主题，在第11页的“描述负载”中。在该示例中，我们还看到了与普通用户相比，名人的写作路径和阅读路径可能会有所不同。在500页之后，我们已经到了整个圈子！

有状态，可离线的客户端

我发现写作和读取路径之间的边界很有趣，因为我们可以讨论移动这个边界并探讨实际意义上的这种转变意味着什么。我们来看看不同情况下的想法。

过去二十年来，Web应用程序的巨大流行使我们对应用程序开发有了一定的假设，这很容易被视为理所当然。特别是，客户机/服务器模型（客户机主要是无状态的，服务器拥有数据的权限）非常普遍，我们几乎忘记了其他任何东西都存在。但是，技术不断发展，我认为不时质疑现状非常重要。

传统上，网络浏览器是无状态的客户端，只有在您连接到互联网时才能做有用的事情（只有您可以离线执行的唯一的事情是在您之前在线加载的页面上下滚动）。然而，最近的“单页面”JavaScript Web应用程序已经获得了很多有状态的功能，包括客户端用户界面交互和Web浏览器中的持久本地存储。移动应用程序可以类似地在设备上存储大量状态，并且不需要往返于大多数用户交互的服务器。

这些不断变化的功能引发了对离线优先应用程序的重新兴趣，这些应用程序尽可能地在同一设备上使用本地数据库，无需连接互联网，并且在网络连接时与后台远程服务器同步可用【42】。由于移动设备通常具有缓慢且不可靠的蜂窝互联网连接，因此，如果用户的用户界面不必等待同步网络请求，并且应用程序大多离线工作，则对用户来说是一大优势（请参阅“具有离线操作的客户端”第170页）。

当我们摆脱无国籍客户与中央数据库交谈的假设，并转向终端用户设备上维护的状态时，开启了一个全新的机会。特别是，我们可以将设备上的状态视为服务器上的状态缓存。屏幕上的像素是客户端应用程序中模型对象的物化视图;模型对象是远程数据中心的本地状态副本【27】。

将状态更改推送给客户端

在典型的网页中，如果您在Web浏览器中加载页面，并且随后服务器上的数据发生更改，则浏览器在重新加载页面之前不会查找有关更改。浏览器只能在一个时间点读取数据，假设它是静态的 —— 它不会订阅来自服务器的更新。因此，设备上的状态是一个陈旧的缓存，除非您明确轮询更改，否则不会更新。（像RSS这样的基于HTTP的订阅源订阅协议实际上只是一种基本的调查形式。）

更新的协议已经超越了HTTP的基本请求/响应模式：服务器发送的事件（EventSource API）和WebSockets提供了通信渠道，通过这些通信渠道，Web浏览器可以与服务器保持开放的TCP连接，服务器可以只要保持连接状态，就会主动将消息推送到浏览器。这为服务器提供了一个机会，主动通知最终用户客户端本地存储状态的任何变化，从而减少客户端状态的陈旧程度。

就我们的写入路径和读取路径模型而言，主动将状态改变到客户端设备意味着将写入路径一直延伸到最终用户。当客户端首次初始化时，它仍然需要使用读取路径来获取其初始状态，但此后可能依赖于服务器发送的状态更改流。我们在流处理和消息传递方面讨论的想法并不局限于仅在数据中心运行：我们可以进一步采用这些想法，并将它们一直延伸到终端用户设备【43】。

这些设备有时会脱机，并且在此期间无法收到服务器状态更改的任何通知。但是我们已经解决了这个问题：在第449页的“消费者偏移量”中，我们讨论了基于日志的消息中介的使用者在失败或断开连接后可以重新连接，并确保它不会错过任何到达的消息它被断开。同样的技术适用于单个用户，每个设备都是小事件的小用户。

端到端的事件流

最近用于开发有状态客户端和用户界面的工具（如Elm语言【30】和Facebook的React，Flux和Redux工具链）已经通过订阅表示用户的事件流来管理内部客户端状态输入或来自服务器的响应，其结构与事件源相似（请参阅第457页的“事件源”）。

将这种编程模型扩展为允许服务器将状态改变事件推送到客户端事件管道中是非常自然的。因此，状态变化可以通过端到端的写入路径流动：从触发状态改变的一个设备上的交互，通过事件日志以及通过多个衍生的数据系统和流处理器，一直到用户界面在另一台设备上观察状态的人。这些状态变化可以以相当低的延迟传播——比如说，在一秒内结束。

一些应用程序（如即时消息传递和在线游戏）已经具有这种“实时”体系结构（从低延迟的交互意义上说，不是“响应时间保证”在本页中的含义）。但为什么我们不用这种方式构建所有的应用程序？

挑战在于无状态客户端和请求/响应交互的假设在我们的数据库，库，框架和协议中非常深入。许多数据存储支持读取和写入操作，请求返回一个响应，但是少得多提供订阅更改的能力 —— 即随着时间的推移返回响应流的请求（请参阅“更改流的API支持”。

为了将写入路径扩展到最终用户，我们需要从根本上重新思考我们构建这些系统的方式：从请求/响应交互转向发布/订阅数据流【27】。我认为更具响应性的用户界面和更好的离线支持的优势将使其值得付出努力。如果您正在设计数据系统，我希望您会记住订阅更改的选项，而不只是查询当前状态。

读也是事件

我们讨论过，当流处理器将衍生数据写入存储（数据库，缓存或索引）时，以及当用户请求查询该存储时，存储将充当写入路径和读取路径之间的边界。该商店允许对数据进行随机访问读取查询，否则这些查询将需要扫描整个事件日志。

在很多情况下，数据存储与流式传输系统是分开的。但请记住，流处理器还需要维护状态以执行聚合和连接（请参阅第472页的“流连接”）。这种状态通常隐藏在流处理器内部，但是一些框架允许它也被外部客户端查询【45】，将流处理器本身变成一种简单的数据库。

我想进一步考虑这个想法。正如到目前为止所讨论的那样，对商店的写入是通过事件日志进行的，而读取是瞬时网络请求，直接进入存储被查询数据的节点。这是一个合理的设计，但不是唯一可能的设计。也可以将读取请求表示为事件流，并通过流处理器发送读取事件和写入事件;处理器通过将读取结果发送到输出流来响应读取事件【46】。

当写入和读取都被表示为事件，并且被路由到同一个流操作符以便处理时，我们实际上是在读查询流和数据库之间执行流表连接。读取事件需要发送到保存数据的数据库分区（请参阅第214页的“请求路由”），就像批处理和流处理器在连接时需要在同一个键上共同输入一样（请参阅“Reduce端连接和分组“）。

服务请求和正在执行的连接之间的这种对应关系是非常重要的【47】。一次性读取请求只是通过连接运算符传递请求，然后立即忘记它;订阅请求是与连接另一端的过去和未来事件的持续连接。

记录读取事件的日志可能对于追踪整个系统中的因果关系和数据来源也有好处：它可以让您在做出特定决策之前重建用户看到的内容。例如，在网上商店，向客户显示的预测出货日期和库存状态可能影响他们是否选择购买物品【4】。要分析此连接，您需要记录用户查询运输和库存状态的结果。

将读取事件写入持久存储器可以更好地跟踪因果关系（请参阅第493页的“顺序事件以捕获因果关系”），但会产生额外的存储和I/O成本。优化这些系统以减少开销仍然是一个开放的研究问题【2】。但是，如果您已经为了操作目的而记录了读取请求，作为请求处理的副作用，将日志作为请求的来源并不是什么大的改变。

多分区数据处理

对于只涉及单个分区的查询，通过流发送查询和收集响应流的努力可能是过度的。然而，这个想法打开了分布式执行复杂查询的可能性，这需要合并来自多个分区的数据，利用流处理器已经提供的消息路由，分区和加入的基础设施。

Storm的分布式RPC功能支持这种使用模式（请参阅第468页的“消息传递和RPC”）。例如，它已被用于计算在Twitter上看到过网址的人数 —— 即，每个人都推送了该网址的跟随者集合【48】。由于Twitter用户组是分区的，因此这种计算需要合并来自多个分区的结果

这种模式的另一个例子是欺诈预防：为了评估特定购买事件是否具有欺诈风险，您可以检查用户的IP地址，电子邮件地址，帐单地址，送货地址等的信誉分数。这些信誉数据库中的每一个都是自身分区的，因此为特定购买事件收集分数需要一系列具有不同分区数据集的联合【49】。

MPP数据库的内部查询执行图具有相似的特征（请参阅第417页的“比较Hadoop与分布式数据库”）。如果您需要执行这种多分区连接，使用提供此功能的数据库可能比使用流处理器实现它更简单。但是，将查询视为流提供了一个选项，可以实现大规模应用程序，这些应用程序可以在传统的现成解决方案的限制下运行。

目标是正确性

对于只读取数据的无状态服务，出现问题时不会造成什么大问题：您可以修复该错误并重新启动服务，并且一切都恢复正常。像数据库这样的有状态的系统并不是那么简单：它们被设计成永远记住事物（或多或少），所以如果出现问题，效果也可能永远持续下去，这意味着它们需要更仔细的思考【50】。

我们希望构建可靠和正确的应用程序（即，即使面对各种故障，其语义也能很好地定义和理解的程序）。大约四十年来，原子性，隔离性和耐久性（第7章）的交易特性一直是构建正确应用的首选工具。但是，这些基础比看起来更弱：例如见证弱隔离级别的混合（请参见“弱隔离级别”（第233页））。

在某些领域，事务被完全抛弃，并被提供更好性能和可伸缩性的模型取代，但是更复杂的语义（例如，请参阅第167页上的“无主复制”）。一致性经常被讨论，但定义不明确（参见第224页的“一致性”和第9章）。有些人主张我们应该“为了更好的可用性而拥抱弱一致性”，而对实际上的实际意义缺乏清晰的认识。

对于如此重要的话题，我们的理解和我们的工程方法是惊人的片状。例如，确定在特定事务隔离级别或复制配置下运行特定应用程序是否安全是非常困难的【51,52】。通常简单的解决方案似乎在并发性低的情况下正常工作，并且没有错误，但是在要求更高的情况下会出现许多细微的错误。

例如，凯尔金斯伯里（Kyle Kingsbury）的杰普森（Jepsen）实验【53】强调了一些产品声称的安全保证与存在网络问题和崩溃时的实际行为之间的明显差异。即使像数据库这样的基础设施产品没有问题，应用程序代码仍然需要正确使用它们提供的功能，如果配置很难理解，这是很容易出错的（这是弱隔离级别，法定配置， 等等）。

如果您的应用程序可以容忍偶尔以不可预测的方式破坏或丢失数据，那么生活就会简单得多，您可能只需横过手指就能逃脱，希望获得最好的效果。另一方面，如果您需要更强的正确性保证，那么可序列化和原子提交就是建立的方法，但是它们是有代价的：它们通常只在单个数据中心（排除地理分布式体系结构）中工作，您可以实现的规模和容错性能。

虽然传统的交易方式并没有消失，但我也相信，在使应用程序正确和灵活地处理错误方面，并不是最后一句话。在本节中，我将提出一些关于数据流架构中正确性的思考方法。

数据库端到端的争论

仅仅因为应用程序使用提供比较强的安全属性的数据系统（例如可序列化的事务），并不意味着应用程序可以保证没有数据丢失或损坏。例如，如果一个应用程序有一个错误导致它写入不正确的数据，或者从数据库中删除数据，那么可序列化的事务不会为你节省。

这个例子可能看起来很无聊，但值得认真对待：应用程序错误发生，人们犯错误。我在第459页的“状态，流和不可变性”中使用了这个例子来支持不可变和只能追加的数据，因为如果删除错误代码的能力来破坏好的数据，更容易从这些错误中恢复数据。

虽然不变性是有用的，但它本身并非万能的。让我们看看可能发生的数据损坏的一个更为简单的例子。

正好执行一次操作

在第476页的“容错”中，我们遇到了一种精确调用一次（或有效一次）语义的想法。如果在处理消息时出现问题，您可以放弃（丢弃消息 —— 即导致数据丢失）或再次尝试。如果再试一次，第一次就有成功的风险，但是你没有发现成功，所以这个消息最终被处理了两次。

处理两次是数据损坏的一种形式：对于相同的服务向客户收费两次（计费太多）或增加计数器两次（夸大一些度量）是不可取的。在这种情况下，正好一次就意味着安排计算，使得最终效果与没有发生错误的情况相同，即使操作实际上由于某种错误而被重试。我们以前讨论过实现这一目标的几种方法。

最有效的方法之一是使幂等操作（参见第478页的“幂等性”）；即确保它具有相同的效果，无论是执行一次还是多次。但是，采取一种不自然是幂等的操作并使其具有幂等性需要付出一定的努力和关注：您可能需要维护一些额外的元数据（例如更新了值的操作ID集合），并在从一个节点到另一个节点（请参阅第295页上的“领导和锁定”）。

重复抑制

除了流处理之外，还需要抑制重复的相同模式出现在许多其他位置。例如，TCP使用数据包上的序列号将它们按正确的顺序排列在收件人处，并确定网络上是否有数据包丢失或重复。任何丢失的数据包都会被重新传输，并且在将数据交给应用程序之前，TCP堆栈会删除任何重复数据包。

但是，此重复抑制仅适用于单个TCP连接的上下文中。假设TCP连接是一个客户端与数据库的连接，并且它正在执行例12-1中的事务。在许多数据库中，事务与客户端连接有关（如果客户端发送了多个查询，数据库就知道它们属于同一个事务，因为它们是在同一个TCP连接上发送的）。如果客户端在发送COMMIT之后但在从数据库服务器回听之前遇到网络中断和连接超时，则不知道事务是否已被提交或中止（图8-1）。

例**12-1** 将资金从一个账户转移到另一个账户的非赦免

```
BEGIN TRANSACTION;
    UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
    UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;
COMMIT;
```

客户端可以重新连接到数据库并重试事务，但现在它在TCP重复抑制的范围之外。由于例12-1中的交易不是幂等的，可能会发生$22而不是所需的$11。因此，尽管例12-1是一个交易原子性的标准例子，但它实际上并不正确，而真正的银行并不像这样工作【3】。

两阶段提交（请参阅第354页上的“原子提交和两阶段提交（2PC）”）协议会破坏TCP连接和事务之间的1:1映射，因为它们必须允许事务协调器在数据库之后重新连接到数据库一个网络故障，并告诉它是否提交或中止有疑问的交易。这足以确保交易只能执行一次吗？不幸的是，即使我们可以抑制数据库客户端和服务器之间的重复事务，我们仍然需要担心最终用户设备和应用程序服务器之间的网络。例如，如果最终用户客户端是Web浏览器，则可能使用HTTP POST请求向服务器提交指令。也许用户处于一个微弱的蜂窝数据连接，他们成功地发送POST，但是信号在他们能够从服务器接收响应之前变得太弱。

在这种情况下，用户可能会显示错误消息，并且可能会手动重试。 Web浏览器警告说，“你确定要再次提交这个表单吗？”—— 用户说是，因为他们希望操作发生。 （Post / Redirect / Get模式【54】可以避免在正常操作中出现此警告消息，但如果POST请求超时，它将无济于事。）从Web服务器的角度来看，重试是一个单独的请求，并且从数据库的角度来看，这是一个单独的事务。通常的重复数据删除机制无济于事。

操作标识符

要通过几次网络通信使操作具有幂等性，仅仅依赖数据库提供的事务机制是不够的—— 您需要考虑请求的端到端流。 例如，您可以为操作（例如UUID）生成唯一的标识符，并将其作为隐藏的表单字段包含在客户机应用程序中，或计算所有相关表单字段的散列以衍生操作ID【3】。如果Web浏览器提交两次POST请求，这两个请求将具有相同的操作ID。然后，您可以将该操作ID传递到数据库，并检查您是否只使用给定的ID执行一个操作，如例12-2中所示。 例**12-2** 使用唯一的**ID**来抑制重复的请求

```
ALTER TABLE requests ADD UNIQUE (request_id);

BEGIN TRANSACTION;
    INSERT INTO requests(request_id, from_account, to_account, amount)
    VALUES('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);
    UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
    UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;
COMMIT;
```

例12-2依赖于 `request_id` 列上的唯一性约束。如果一个事务尝试插入一个已经存在的ID，那么 `INSERT` 失败，事务被中止，使其无法生效两次。即使在较弱的隔离级别下，关系数据库也能正确地维护唯一性约束（而在第248页上的“写入偏差和幻读”中讨论过，应用程序级别的check-then-insert可能会在不可序列化的隔离下失败）。

除了抑制重复的请求之外，示例12-2中的请求表充当事件日志的一种，暗示着事件源的方向（请参阅第457页的“事件源”）。账户余额的更新事实上不必与插入事件相同的事务发生，因为它们是多余的，并且可以从下游消费者中的请求事件衍生出来 —— 只要该事件只处理一次，这可以再次使用请求ID来执行。

端到端的论点

抑制重复交易的这种情况只是一个更普遍的原则的一个例子，这个原则被称为端对端的论点，它在1984年由Saltzer，Reed和Clark阐述【55】：

只有在通信系统端点的应用程序的知识和帮助下，所讨论的功能才能够完全正确地实现。因此，将这种被质疑的功能作为通信系统本身的功能是不可能的。（有时，通信系统提供的功能的不完整版本可能有助于提高性能。）

在我们的例子中，有问题的函数是重复抑制。我们看到TCP在TCP连接级别抑制重复的数据包，一些流处理器在消息处理级别提供了所谓的唯一的语义，但是这不足以防止用户提交重复请求一次。 TCP，数据库事务和流处理器本身并不能完全排除这些重复。解决这个问题需要一个端到端的解决方案：从最终用户客户端一直传递到数据库的事务标识符。

端到端参数也适用于检查数据的完整性：以太网，TCP和TLS中内置的校验和可以检测网络中数据包的损坏情况，但是它们无法检测到发送和接收软件中的错误网络连接的末端，或数据存储在磁盘上的损坏。如果您想要捕获所有可能的数据损坏源，则还需要端到端的校验和。

类似的说法适用于加密【55】：家庭WiFi网络上的密码可以防止人们窃听您的WiFi流量，但不会对互联网上其他地方的攻击者进行窥探;您的客户端和服务器之间的TLS / SSL可以得到保护对网络攻击者，而不是对服务器的妥协。只有端到端的加密和认证可以防止所有这些事情。

尽管低级功能（TCP复制抑制，以太网校验和，WiFi加密）无法单独提供所需的端到端功能，但它们仍然很有用，因为它们可以降低较高级别出现问题的可能性。例如，如果我们没有TCP将数据包放回正确的顺序，那么HTTP请求通常会被破坏。我们只需要记住，低级别的可靠性功能本身并不足以确保端到端的正确性。

在数据系统中应用端到端思考

这使我回到我的原始论文：仅仅因为应用程序使用提供比较强的安全属性的数据系统（如可序列化事务），并不意味着应用程序保证不会丢失数据或损坏。应用程序本身也需要采取端到端的措施，例如重复压制。

这是一个耻辱，因为容错机制很难得到正确的。低级可靠性机制（如TCP中的可靠性机制）运行良好，因此其余较高级别的故障发生得相当少。将抽象中的高级容错机制封装起来非常好，以便应用程序代码不必担心它 —— 但是我担心我们还没有找到合适的抽象。

长期以来，交易被认为是一个很好的抽象，我相信它们是有用的。正如第7章介绍中所讨论的那样，它们会涉及各种可能的问题（并发写入，约束违规，崩溃，网络中断，磁盘故障），并将其折叠为两种可能的结果：提交或中止。这是编程模型的一个巨大的简化，但我担心这是不够的。

事务处理非常昂贵，尤其是涉及异构存储技术时（请参阅第364页的“实践中的分布式事务”）。当我们拒绝使用分布式事务是因为它们太昂贵时，我们最终不得不在应用程序代码中重新实现容错机制。正如本书中大量的例子所显示的那样，关于并发性和部分失败的推理是困难且违反直觉的，所以我怀疑大多数应用程序级别的机制不能正常工作。结果是丢失或损坏的数据。

出于这些原因，我认为值得探索的容错抽象方法能够容易地提供特定于应用程序的端到端正确性属性，而且还可以在大型分布式环境中保持良好的性能和良好的操作特性。

强制实施约束

让我们考虑非捆绑数据库的想法背景下的正确性（“剥离数据库”，第499页）。我们看到，使用从客户端一直传递到记录写入的数据库的请求标识可以实现端到端的重复压缩。其他类型的限制呢？

特别是让我们关注唯一性约束 —— 例如我们在例12-2中所依赖的约束。在第330页的“约束和唯一性保证”中，我们看到了几个其他需要强制实施唯一性的应用程序功能示例：用户名或电子邮件地址必须唯一标识用户，文件存储服务不能包含多个文件同名，两个人不能在航班或剧院预订同一个座位。

其他类型的约束非常相似：例如，确保帐户余额永远不会变为负数，您不会出售比仓库中的库存更多的物料，或者会议室没有重复的预订。执行唯一性的技术通常也可以用于这些约束。

唯一性约束需要达成共识

在第9章中我们看到，在分布式环境中，强制执行唯一性约束需要达成共识：如果存在多个具有相同值的并发请求，则系统需要决定哪个冲突操作被接受，并拒绝其他违规操作的约束。

达成这一共识的最常见方式是将单个节点作为领导者，并将其负责制定所有决策。只要您不介意通过单个节点发送所有请求（即使客户端位于世界的另一端），并且只要该节点没有失败，就可以正常工作。如果您需要容忍领导者失败，那么您又回到了共识问题（请参阅第367页上的“单领导表示和共识”）。

唯一性检查可以根据需要唯一的值进行划分。例如，如果需要通过请求标识确保唯一性（如例12-2所示），则可以确保具有相同请求标识的所有请求都路由到同一分区（请参阅第6章）。如果您需要用户名是唯一的，您可以通过用户名的哈希分区。

但是，排除了异步多主复制，因为它可能会导致不同的主设备同时接受冲突的写操作，因此这些值不再是唯一的（请参阅第295页的“实现可线性化系统”）。如果你想立即拒绝任何违反约束的写入，同步协调是不可避免的【56】。

基于日志的消息传递的唯一性

该日志确保所有消费者以相同的顺序查看消息 —— 这种保证在形式上被称为全部命令广播并且等同于共识（参见第346页上的“全序广播”）。在使用基于日志的消息传递的非捆绑数据库方法中，我们可以使用非常类似的方法来执行唯一性约束。

流处理器在单个线程上依次占用日志分区中的所有消息（请参见第448页的“与传统消息传递相比的日志”）。因此，如果日志根据需要唯一的值进行分区，则流处理器可以明确并确定性地确定几个冲突操作中的哪一个先到达。例如，在多个用户尝试声明相同用户名的情况下【57】：

1. 对用户名的每个请求都被编码为一条消息，并附加到由用户名散列确定的分区。
2. 流处理器使用本地数据库连续读取日志中的请求，以跟踪使用哪些用户名。对于每个可用的用户名请求，它都会记录该名称并将成功消息发送到输出流。对于每个已经被使用的用户名请求，它都会向输出流发送拒绝消息。
3. 请求用户名的客户端观察输出流并等待与其请求相对应的成功或拒绝消息。

该算法基本上与第363页上的“使用全序广播实现线性化存储”中的算法相同。它可以通过增加分区数容易地扩展为较大的请求吞吐量，因为可以独立处理每个分区。

该方法不仅适用于唯一性约束，而且适用于许多其他类型的约束。其基本原理是任何可能冲突的写入都会路由到相同的分区并按顺序处理。正如第174页上的“什么是冲突？”和第246页上的“写入偏差和幻读”中所述，冲突的定义可能取决于应用程序，但流处理器可以使用任意逻辑来验证请求。这个想法与Bayou在20世纪90年代开创的方法类似【58】。

多分区请求处理

当涉及多个分区时，确保操作以原子方式执行，同时满足约束条件变得更有趣。在示例12-2中，可能有三个分区：一个包含请求ID，一个包含收款人账户，另一个包含付款人账户。没有理由把这三件事放在同一个分区，因为它们都是相互独立的。

在数据库的传统方法中，执行此事务需要跨所有三个分区进行原子提交，这实质上会将它强制为与任何这些分区上的所有其他事务的总顺序。由于现在存在跨分区协调，不能再独立处理不同的分区，因此吞吐量可能会受到影响。 但是，事实证明，使用分区日志可以实现同等的正确性，并且不需要原子提交：

1. 从账户A向账户B转账的请求由客户端提供唯一的请求ID，并根据请求ID附加到日志分区。
2. 流处理器读取请求的日志。对于每个请求消息，它发出两条消息以输出流：付款人账户A（由A分配）的借方指令和收款人账户B（由B分区）的信贷指令。原始的请求ID包含在那些发出的消息中。
3. 其他处理器使用信用卡和借记指令流，通过请求ID进行扣除，并将更改应用于账户余额。

步骤1和步骤2是必要的，因为如果客户直接发送信用和借记指令，则需要在这两个分区之间进行原子提交以确保两者都不发生。为了避免分布式事务的需要，我们首先将请求永久记录为单条消息，然后从第一条消息中获取信用和借记指令。单对象写入在几乎所有数据系统中都是原子性的（请参阅“单对象写入”第213页），因此请求既可以出现在日志中，也可以不出现，而不需要多分区原子com-麻省理工学院。

如果步骤2中的流处理器崩溃，则从上一个检查点恢复处理。这样做时，它不会跳过任何请求消息，但可能会多次处理请求并产生重复的信用和借记指令。但是，由于它是确定性的，因此它只会再次生成相同的指令，并且步骤3中的处理器可以使用端到端请求ID轻松地对它们进行重复数据删除。

如果您想确保付款人帐户不会因此次转账而透支，您可以额外使用流处理器（分区）使用多个不同分区的阶段的想法与我们所讨论的类似“多分区数据处理”一节第514页（另请参阅“并发控制”一节第462页）。

及时性与完整性

事务的一个方便属性是它们通常是可线性化的（请参阅“可用性”），也就是说，一个作者等待事务提交，之后其所有读者立即可以看到它的写入。

在跨流处理器的多个阶段拆分操作时情况并非如此：日志的使用者在设计上是异步的，因此发送者不会等到其消息已经被消费者处理。但是，客户端可能会等待消息出现在输出流上。这是我们在检查是否满足唯一性约束时在“基于日志的消息传递中的唯一性”一节中所做的操作。

在这个例子中，唯一性检查的正确性不取决于消息的发送者是否等待结果。等待仅具有同步通知发送者唯一性检查是否成功的目的，但是该通知可以与处理消息的效果分离。

更一般地说，我认为术语一致性这个术语将两个不同的需求分开考虑：

及时性

及时性意味着确保用户观察系统处于最新状态。我们之前看到，如果用户从数据的陈旧副本中读取数据，他们可能会以不一致的状态观察数据（请参阅第161页上的“复制延迟问题”）。但是，这种不一致是暂时的，最终只能通过等待和再次尝试来解决。

CAP定理（参见第359页的“线性一致性的代价”）使用线性一致的意义上的一致性，这是实现及时性的强有力的方法。像写后读一致性这样的时效性较弱的属性（请参阅第162页的“读己之写”）也很有用。

完整性 诚信意味着没有腐败;即没有数据丢失，并且没有矛盾或错误的数据。尤其是，如果将某些衍生数据集作为某些基础数据的视图进行维护（请参阅“从事件日志导出当前状态”（第458页）），衍生必须正确。例如，数据库索引必须正确地反映数据库的内容 —— 缺少某些记录的索引不是很有用。如果完整性受到侵犯，这种不一致是永久性的：在大多数情况下，

等待并再次尝试不会修复数据库损坏。相反，需要明确的检查和修理。在ACID事务的上下文中（参见第223页上的“ACID的含义”），一致性通常被理解为某种特定于应用程序的完整性概念。原子性和耐久性是保持完整性的重要工具。

口号形式：违反及时性是“最终一致性”，而违反诚信则是“永久不一致”。

我要断言，在大多数应用中，完整性比时间要重要得多。违反时效可能令人讨厌和混淆，但是对正直的侵犯可能是灾难性的。

例如，在您的信用卡对账单上，如果您在过去24小时内完成的交易尚未出现，这并不奇怪 —— 这些系统有一定的滞后是正常的。我们知道银行协调和异步结算交易，并且这里的及时性并不重要【3】。但是，如果报表余额不等于交易总和加上先前的报表余额（数额错误），或者交易是向您收取但未支付给商家的话，那将是非常糟糕的（消失的金钱）。这样的问题会违反系统的完整性。

数据流系统的正确性

ACID事务通常既提供时间性（例如线性化）又提供完整性（例如原子提交）保证。因此，如果从ACID交易的角度来看应用程序的正确性，那么时间性与完整性的区别就相当不重要了。

另一方面，我们在本章中讨论的基于事件的数据流系统的一个有趣特性是它们将时间性和完整性分开。在异步处理事件流时，不能保证及时性，除非在返回之前明确地构建等待消息到达的消费者。但是，完整性实际上是流式传输系统的核心。

一次或一次有效的语义（请参阅“故障容错”一节第437页）是一种保持完整性的机制。如果事件丢失，或者事件发生两次，数据系统的完整性可能被侵犯。因此，容错消息传递和重复抑制（例如，幂等操作）对于在面对故障时保持数据系统的完整性是重要的。

正如我们在上一节看到的那样，可靠的流处理系统可以在不需要分布式事务和原子提交协议的情况下保持整体性，这意味着它们可以实现更好的可比较的正确性

性能和运行稳健性。我们通过以下机制的结合实现了这一完整性：

- 将写入操作的内容表示为单个消息，可以轻松地以原子方式编写 —— 这种方法非常适合事件采购（请参阅第457页的“事件采购”）。
- 使用确定性描述函数从该单个消息中获取所有其他状态更新，这与存储过程类似（请参见第252页的“真的串行执行”和第479页的“作为衍生函数的应用程序代码”）
- 通过所有这些级别的处理传递客户端生成的请求ID，启用端到端重复抑制和幂等性
- 使消息不可变并允许衍生数据不时重新处理，这使得从错误中恢复变得更加容易（请参阅“不可变事件的优点”第367页）

这种机制的组合在我看来是未来构建容错应用程序的一个非常有前途的方向。

松散的解释约束

如前所述，执行唯一性约束需要达成共识，通常通过在单个节点中汇集特定分区中的所有事件来实现。如果我们想要传统的唯一性约束形式，并且流处理无法避免，这种限制是不可避免的。

然而，另一个要认识到的是，许多真正的应用程序实际上可以摆脱唯一性较弱的概念：

- 如果两个人同时注册相同的用户名或预订相同的座位，则可以发送其中一个消息来道歉，并要求他们选择不同的用户名。这种纠正错误的变化被称为补偿性事务【59,60】。
- 如果客户订购的物品多于仓库中的物品，则可以订购更多库存，为延误向客户道歉，并向他们提供折扣。实际上，如果叉车在仓库中的某些物品上方跑过来，而库存的物品数量比您想象的要少，那么您就必须这样做【61】。因此，无论如何，道歉工作流程已经需要成为业务流程的一部分，因此可能不需要对库存中的项目数量进行线性化约束。
- 同样地，许多航空公司预计飞行员会错过飞机，许多旅馆超额预订客房，预计部分客人将取消预订。在这些情况下，出于商业原因故意违反了“每个座位一人”的约束，并且处理补偿过程（退款，升级，在邻近酒店提供免费房间）以处理需求超过供应的情况。即使没有超额预订，为了应对由于恶劣天气而被取消的航班或者罢工的员工，这些问题的恢复仅仅是商业活动的正常组成部分，就需要道歉和赔偿流程。
- 如果有人收取比他们账户中更多的钱，银行可以向他们收取透支费用，并要求他们偿还欠款。通过限制每天的提款总额，银行的风险是有限的。

在许多商业环境中，临时违反约束并稍后通过道歉修复它实际上是可以接受的。道歉的成本（金钱或报酬）各不相同，但通常很低：您无法取消发送电子邮件，但可以发送后续电子邮件并进行更正。如果您不小心向信用卡收取了两次费用，您可以退还其中一项费用，而您的费用仅仅是处理费用，并且可以处理客户投诉。一旦自动提款机支付了钱，就不能直接将其退回，尽管原则上如果账户透支并且客户不支付，原则上可以派遣收债员收回款项。

道歉的成本是否可以接受是一个商业决策。如果可以接受的话，在写入数据之前检查所有约束的传统模型是不必要的限制，并且不需要线性化约束。乐观地继续写作，并在事实之后检查约束，这可能是一个合理的选择。您仍然可以确保验证发生在做恢复成本高昂的事情之前，但这并不意味着在您编写数据之前您必须先进行验证。

这些应用程序确实需要完整性：您不会希望失去预订，或者由于信用和借方不匹配而导致资金消失。但是他们并不需要及时执行约束：如果您销售的物品多于仓库中的物品，则可以在事后道歉后修补问题。这样做与我们在第171页上的“处理写入冲突”中讨论的冲突解决方法类似。

协调避免数据系统

我们现在做了两个有趣的观察：

1. 数据流系统可以保持对衍生数据的完整性保证，而无需原子提交，线性化或同步跨分区协调。
2. 虽然严格的唯一性约束要求及时性和协调性，但许多应用程序实际上可以很好地处理宽松的约束，只要整个过程保持完整性，它们可能会被暂时违反并予以修复。

总之，这些观察意味着数据流系统可以为许多应用程序提供数据管理服务，而不需要协调，同时仍然提供强大的完整性保证。这种避免协调的数据系统具有很大的吸引力：它们可以比需要执行同步协调的系统获得更好的性能和容错能力【56】。

例如，这种系统可以在多引导器配置中跨多个数据中心进行分布式操作，在区域之间异步复制。任何一个数据中心都可以继续独立运行，因为不需要同步跨区域协调。这样一个系统的时效性保证会很弱 —— 如果不引入协调，它就不可能线性化，但它仍然可以提供强有力的完整性保证。

在这种情况下，序列化事务作为维护衍生状态的一部分仍然是有用的，但它们可以在小范围内运行，在那里它们工作得很好【8】。异构分布式事务（如XA事务）（请参阅“实践中的分布式事务”（第367页））不是必需的。同步协调仍然可以在需要的地方引入（例如，在无法恢复的操作之前执行严格的限制），但是如果只有一个小的协议，则不需要任何东西来支付协调费用应用程序的一部分需要它【43】。

查看协调和约束的另一种方法是：它们减少了由于不一致而必须做出的道歉数量，但也可能会降低系统的性能和可用性，从而可能会增加必须制定的道歉数量中断。您不能将道歉数量减少到零，但您可以根据自己的需求寻找最佳平衡点 —— 这是既不存在太多不一致性又不存在太多可用性问题的最佳选择。

信任，但需要验证

我们所有关于正确性，完整性和容错性的讨论都假设某些事情可能会出错，但其他事情则不会。我们将这些假设称为我们的系统模型（请参阅“将系统模型映射到现实世界”一节第309页）：例如，我们应该假设进程可能会崩溃，机器可能突然断电，网络可能会任意延迟或丢弃消息。但是我们也可以假设写入磁盘的数据在`fsync`之后不会丢失，内存中的数据没有损坏，并且CPU的乘法指令总是返回正确的结果。

这些假设是非常合理的，因为大多数时候它们都是真实的，如果我们不得不经常担心计算机出错，那么很难完成任何事情。传统上，系统模型采用二元方法处理故障：我们假设有些事情会发生，而其他事情永远不会发生。实际上，这更像是一个概率问题：有些事情更可能，其他事情不太可能。问题是违反我们的假设是否经常发生，以至于我们可能在实践中遇到它们。

我们已经看到，数据可能会在磁盘未触及时破损（请参阅第227页的“复制和耐久性”），并且网络上的数据损坏有时可能会妨碍TCP校验和（请参阅第293页的“虚弱形式的说谎”）。也许这是我们应该更加关注的事情？

我过去曾经使用过的一个应用程序收集了来自客户端的崩溃报告，我们收到的一些报告只能通过在这些设备的内存中随机位翻转来解释。这看起来不太可能，但是如果你有足够的设备来运行你的软件，那么即使发生的事情也不会发生。除了由于硬件故障或辐射导致的随机存

储器损坏之外，某些病态存储器访问模式甚至可以在没有故障的存储器中翻转位【62】——可用于破坏操作系统中安全机制的效应【63】技术被称为rowhammer）。一旦你仔细观察，硬件并不是完美的抽象。

要清楚的是，随机位翻转在现代硬件上仍然非常罕见【64】。我只想指出，他们并没有超越可能性领域，所以他们值得关注。

在软件缺陷面前保持完整性

除了这些硬件问题之外，总是存在软件错误的风险，这些错误不会被较低级别的网络，内存或文件系统校验和所捕获。即使广泛使用的数据库软件也存在一些问题：即使MySQL和PostgreSQL是健壮且充分的，但我个人发现MySQL的例子未能正确维护唯一性约束【65】和PostgreSQL的序列化隔离级别，表现出写入偏斜异常【66】他们认为这些数据库已经被许多人进行了多年的战斗测试。在不太成熟的软件中，情况可能会更糟糕。

尽管在仔细设计，测试和审查方面做出了相当大的努力，但错误仍在蔓延。虽然它们很少，并且最终被发现并修复，但仍然存在一段时期，这些错误可能会破坏数据。

当涉及到应用程序代码时，我们不得不承担更多的错误，因为大多数应用程序在数据库代码所做的评审和测试的数量上没有接近的地方。许多应用程序甚至没有正确使用数据库提供的用于保存完整性的功能，例如外键或唯一性约束【36】。

ACID意义上的一致性（请参阅第224页的“一致性”）基于数据库以一致状态启动并且事务将其从一个一致状态转换为另一个一致状态的想法。因此，我们期望数据库始终处于一致状态。但是，如果您认为该交易没有错误，则这个概念才有意义。如果应用程序错误地使用数据库以某种方式，例如，不安全地使用弱隔离级别，数据库的完整性无法得到保证。

不要盲目信任他们的承诺

由于硬件和软件并不总是符合我们希望的理想，所以数据腐败似乎迟早是不可避免的。因此，我们至少应该有办法查明数据是否已经损坏，以便我们能够修复它并尝试追查错误的来源。检查数据的完整性称为审计。

如“不可变事件的优点”一节中所述，审计不仅适用于财务应用程序。然而，可审计性在财务中非常重要，因为每个人都知道错误发生了，而且我们都认识到需要能够检测和解决问题。

成熟的系统同样倾向于考虑不太可能的事情出错的可能性，并管理这种风险。例如，HDFS和Amazon S3等大规模存储系统不完全信任磁盘：它们运行后台进程，这些后台进程持续读回文件，将其与其他副本进行比较，并将文件从一个磁盘移动到另一个磁盘，以便减轻沉默腐败的风险【67】。

如果你想确保你的数据仍然存在，你必须真正阅读并检查。大多数时候它仍然会在那里，但如果不是这样，你真的很想早日找到答案。通过同样的说法，尝试从备份中不时恢复是非常重要的，否则只有当它太迟而您已经丢失数据时才会发现备份被破坏。不要盲目地相信它是全部工作。

验证的文化

像HDFS和S3这样的系统仍然必须假定磁盘大部分时间都能正常工作 —— 这是一个合理的假设，但假设它们始终正常工作并不相同。然而，目前还没有多少系统有这种“信任但是验证”的方式来持续审计自己。许多人认为正确性保证是绝对的，并且没有为罕见数据损坏的可能性做出规定。我希望将来我们会看到更多的自我验证或自我审计系统，不断检查自己的诚信，而不是依赖盲目的信任【68】。

我担心ACID数据库的文化导致我们在盲目信任技术（如交易机制）的基础上开发应用程序，而忽视了这种过程中的任何可审计性。由于我们所信任的技术在大多数情况下工作得很好，审计机制并不值得投资。

但随之而来的是数据库的格局发生了变化：在NoSQL的旗帜下，一致性的保证变得越来越少，成熟的存储技术越来越被广泛的使用。但是，由于审计机制还没有形成，我们继续在盲目信任的基础上建立应用程序，即使这种方法现在变得更加危险。让我们思考一下关于可审计性的设计。

设计可审计性

如果一个事务在一个数据库中改变了多个对象，事实上很难说事实是什么意思。即使您捕获事务日志（请参阅第454页上的“更改数据捕获”），各种表中的插入，更新和删除操作并不一定清楚表明为何执行这些突变。决定这些突变的应用逻辑的调用是暂时的，不能被复制。

相比之下，基于事件的系统可以提供更好的可审计性。在事件源方法中，系统的用户输入被表示为一个单一的不可变事件，并且任何结果状态更新都来自该事件。衍生可以做出确定性和可重复性，以便通过相同版本的衍生代码运行相同的事件日志将导致相同的状态更新。

清楚地看到数据流（请参阅第419页上的“批处理输出的原理”）可以使数据的来源更加清晰，从而使完整性检查更加可行。对于事件日志，我们可以使用散列来检查事件存储没有被破坏。对于任何衍生状态，我们可以重新运行从事件日志中衍生它的批处理和流处理器，以检查是否获得相同的结果，或者甚至并行运行冗余衍生。

确定性和定义良好的数据流也使调试和跟踪系统的执行变得容易，以确定它为什么做了某些事情【4,69】。如果出现意想不到的事情，那么具有诊断能力来重现导致意外事件的确切环境 —— 一种时间行程调试功能是非常有价值的。

再次是端到端的论点

如果我们不能完全相信系统的每个组件都不会发生腐败 —— 每一个硬件都是无错的，并且每一个软件都没有缺陷 —— 那么我们至少必须定期检查数据的完整性。如果我们不检查，我们就不会发现腐败，直到它太晚了，并且已经造成了一些下游损害，在这一点上追踪这个问题将变得更加困难和昂贵。

检查数据系统的完整性最好是以端到端的方式完成（请参阅“数据库的端到端争论”，51页）：我们可以在完整性检查中包含的系统越多，那里的机会就越少在这个过程的某个阶段，腐败是被忽视的。如果我们可以检查整个衍生数据管道是端到端正确的，那么沿着路径的任何磁盘，网络，服务和算法都隐含在检查中。

持续的端到端完整性检查可以提高您对系统正确性的信心，从而使您的移动速度更快【70】。与自动化测试一样，审计增加了发现错误的可能性，从而降低了系统更改或新存储技术造成损害的风险。如果您不害怕进行更改，您可以更好地开发应用程序以满足不断变化的需求。

审计数据系统的工具

目前，数据系统并不多，这使可审计性成为高层关注的重点。一些应用程序实现自己的审计机制，例如将所有更改记录到单独的审计表中，但是保证审计日志的完整性，并且数据库状态仍然很困难。事务日志可以通过定期使用硬件安全模块对事务日志进行签名来进行防篡改，但这并不能保证正确的事务首先进入日志。

使用加密工具来证明系统的完整性是十分有趣的，这种方式对于广泛的硬件和软件问题以及甚至是潜在的恶意行为都很有效。加密货币，区块链和分布式账本技术，如比特币，以太坊，波纹，恒星等等【71,72,73】已经出现在这个领域。

我没有资格评论这些技术作为商定合同的货币或机制的优点。但是，从数据系统的角度来看，它们包含了一些有趣的想法。实质上，它们是分布式数据库，具有数据模型和事务机制，不同副本可以由互不信任的组织托管。复制品不断检查彼此的完整性，并使用共识协议来约定应执行的交易。

我对这些技术的拜占庭容错方面有些怀疑（参见第304页的“拜占庭故障”），而且我发现工作证明（比如比特币挖掘）技术非常浪费。比特币的交易吞吐量相当低，尽管出于政治和经济原因而不是技术交易。但是，完整性检查方面很有趣。

我可以想象完整性检查和审计算法（如证书透明度和分布式分类账算法）在一般数据系统中得到越来越广泛的应用。 一些工作将需要使它们具有同样的可扩展性，因为没有加密审计的系统，并且尽可能降低性能损失。 但我认为这是一个值得关注的领域。

做正确的事情

在本书的最后部分，我想退后一步。在本书中，我们考察了各种不同的数据系统体系结构，评估了它们的优缺点，并探讨了构建可靠，可扩展和可维护应用程序的技术。但是，我们忽略了现在我想填补的讨论中一个重要的基本部分。

每个系统都是为了一个目的而建造的我们采取的每一项行动都有既定的意义，也有无意义的后果。其目的可能与赚钱一样简单，但世界的后果可能会远远超出最初的目的。我们这些建立这些系统的工程师有责任仔细考虑这些后果并有意识地决定我们希望生活在哪一种世界。

我们将数据作为一个抽象的东西来讨论，但请记住，许多数据集都是关于人的：他们的行为，他们的兴趣，他们的身份。我们必须以人性和尊重来对待这些数据。用户也是人类，人的尊严是至高无上的。

软件开发日益涉及重要的道德选择。有一些指导原则可以帮助软件工程师解决这些问题，例如ACM的软件工程道德规范和专业实践【77】，但很少在实践中讨论，应用和实施。因此，工程师和产品经理有时会对隐私和产品潜在的负面后果采取非常傲慢的态度【78,79,80】。

技术本身并不好或不好，重要的是如何使用它以及它如何影响人。对于像搜索引擎这样的软件系统来说，就像使用像武器这样的武器非常相似。我认为，软件工程师仅仅专注于技术并忽视其后果是不够的：伦理责任也是我们的责任。有关道德的推理是困难的，但忽视这一点太重要了。

预测分析

例如，预测分析是“大数据”炒作的主要部分。使用数据分析预测天气或疾病传播是一回事【81】;另一个问题是要预测一个罪犯是否可能再犯罪，一个贷款申请人是否有可能违约，或者一个保险客户是否有可能做出昂贵的索赔。后者直接影响到个人的生活。当然，支付网络希望防止欺诈交易，银行希望避免不良贷款，航空公司希望避免劫持，公司希望避免激怒无效或不可靠的人。从他们的角度来看，错过商业机会的成本很低，但不良贷款或有问题的员工的成本要高得多，因此组织希望保持谨慎是很自然的。如果有疑问，他们最好不要说。

然而，随着算法决策变得越来越普遍，某人（准确地或错误地）被某种算法标记为有风险，可能会遭受大量“不”决定。系统地排除工作，航空旅行，保险，物业租赁，金融服务和社会其他关键方面是个人自由的巨大制约因素，因此被称为“算法监狱”【82】。在尊重人权的国家，刑事司法系统在被证明有罪之前就认定无罪。另一方面，自动化系统可以有系统地，任意地排除一个人参与社会活动，而没有任何内疚的证据，而且几乎没有上诉的机会。

偏见和歧视

算法做出的决定不一定比人类做出的更好或更差。每个人都可能有偏见，即使他们积极地试图抵消它们，歧视性做法也可能在文化上被制度化。人们希望根据数据做出决定，而不是通过人们的主观和现场评估来更加公平，给那些在传统系统中经常被忽视的人更好的机会【83】。

当我们开发预测分析系统时，我们不仅仅是通过使用软件来指定什么时候说是或否的规则来自动化人的决定;我们甚至将规则本身从数据中推断出来。但是，这些系统学到的模式是不透明的：即使数据中存在一些相关性，我们也可能不知道为什么。如果在算法输入中存在系统偏差，系统很可能会在输出中学习和放大偏差【84】。

在许多国家，反歧视法律禁止根据种族，年龄，性别，性别，残疾或信仰等受保护特征对待不同的人。一个人的数据的其他特征可能会被分析，但是如果他们与受保护的特征相关会发生什么？例如，在种族隔离的邻居中，一个人的邮政编码，甚至是他们的IP地址，都是一个

强烈的种族预测。这样说，相信一种算法可以以某种方式将偏倚的数据作为输入并产生公平和公正的输出【85】，这似乎是荒谬的。然而，这种观点似乎常常被数据驱动型决策制定的支持者所暗示，这种态度被讽刺为“机器学习就像洗钱对于偏见”【86】。

预测分析系统只是从过去推断出来的;如果过去是歧视性的，他们就会把这种歧视归为法律。如果我们希望未来比过去更好，那么就需要道德想象力，而这只有人类才能提供【87】。数据和模型应该是我们的工具，而不是我们的主人。

责任和问责

自动决策开启了责任和问责的问题【87】。如果一个人犯了错误，他们可以被追究责任，受决定影响的人可以上诉。算法也会犯错误，但是如果他们出错，谁会犯错误【88】？当一辆自驾车引发事故时，谁负责？如果自动信用评分算法系统地区分特定种族或宗教的人，是否有任何追索权？如果机器学习系统的决定受到司法审查，您能向法官解释算法是如何做出决定的吗？

信用评级机构是收集数据来做出人们决策的一个老例子。不良信用评分会使生活变得困难，但至少一个信用评分通常是基于一个人的实际借款历史的相关事实，并且记录中的任何错误都可以得到纠正（尽管机构通常不会这么容易）。然而，基于机器学习的评分算法通常使用更广泛的输入范围，并且更加不透明，使得难以理解特定决策是如何发生的以及是否有人正在以不公平或歧视的方式被对待【89】。

信用评分总结了“你过去的表现如何？”，而预测分析通常是基于“谁与你类似，以及过去人们喜欢你的行为如何？”。与他人行为相似的绘图意味着刻板印象人们，例如根据他们居住的地方（一个关于种族和社会经济阶层的密切代理人）。那些放错人的人呢？而且，如果由于错误的数据而做出的决定是不正确的，则追索权几乎是不可能的【87】。

很多数据本质上是统计的，这意味着即使概率分布总体上是正确的，个别情况也可能是错的。例如，如果贵国的平均寿命是80岁，那么这并不意味着在80岁生日时就会死亡。从平均分布和概率分布来看，你不能说很多关于某个特定人的生活年龄。同样，预测系统的输出是概率性的，在个别情况下可能是错误的。

盲目相信数据至高无上的决策不仅是妄想，它是非常危险的。随着数据驱动的决策变得越来越普遍，我们需要弄清楚如何使算法负责任和透明，如何避免加强现有的偏见，以及如何在不可避免的错误时加以修复。

我们还需要弄清楚如何防止数据被用来伤害人，并且实现其正向潜力。例如，分析可以揭示人们生活的财务和社会特征。一方面，这种权力可以用来把援助和支持集中在帮助那些最需要援助的人身上。另一方面，它有时被掠夺性企业用于识别弱势群体并向高风险产品销售，如高成本贷款和无价值的大学学位【87,90】。

反馈回路

即使预测性应用程序对人们的影响较小，比如推荐系统，也存在难以解决的问题。当服务善于预测用户想要看到什么内容时，他们最终可能只会向人们展示他们已经同意的观点，导致产生陈腔滥调，错误信息和极化可能滋生的回声室。我们已经看到社交媒体呼应室对竞选活动的影响【91】。

当预测分析影响人们的生活时，特别是由于自我强化反馈循环而出现有害问题。例如，考虑雇主使用信用评分来评估潜在员工的情况。你可能是一个信誉好的好员工，但是由于你不能控制的不幸，你会突然发现自己陷入财务困境。由于您错过了账单付款，您的信用评分会受到影响，您将不太可能找到工作。失业使你陷入贫困，这进一步恶化了你的分数，使其更难找到工作【87】。由于有毒的假设，这是一个下降的螺旋，隐藏在数学严谨和数据伪装的背后。

我们不能总是预测这种反馈循环何时发生。然而，通过考虑整个系统（不仅仅是计算机化的部分，而且还有与之互动的人），可以预测许多后果—— 一种称为系统思维的方法【92】。我们可以尝试理解数据分析系统如何响应不同的行为，结构或特性。该系统是否加强和扩大了人们之间现有的差异（例如，使富人更穷或更穷），还是试图打击不公正？即使有最好的意图，我们也必须小心意想不到的后果。

隐私和跟踪

除了预测分析的问题 —— 即使用数据来做出关于人的自动决策 —— 数据收集本身也存在道德问题。收集数据的组织与正在收集数据的人之间有什么关系？

当系统仅存储用户明确输入的数据时，系统会以特定方式存储和处理数据，系统正在为用户执行服务：用户就是客户。但是，当用户的活动被跟踪并记录为他们正在做的其他事情的副作用时，这种关系就不那么清晰了。该服务不再仅仅是用户告诉它做的事情，而是服务于它自己的利益，这可能与用户的兴趣相冲突。

跟踪行为数据对于许多在线服务的面向用户的功能变得越来越重要：跟踪点击哪些搜索结果有助于提高搜索结果的排名;推荐“喜欢X的人也喜欢Y”，可以帮助用户发现有趣而有用的东西; A / B测试和用户流量分析可以帮助指出如何改进用户界面。这些功能需要一定量的用户行为跟踪，用户可以从中受益。

但是，根据公司的商业模式，追踪往往不止于此。如果服务是通过广告获得资金支持的，那么广告客户就是真正的客户，而用户的利益则位居第二。跟踪数据变得更加详细，分析变得更深入，数据保留很长时间，以便为营销目的建立每个人的详细资料。

现在，公司和收集数据的用户之间的关系开始变得非常不同。用户可获得免费服务，并尽可能使用户接受该服务。对用户的追踪主要不是服务于该个人，而是服务于该服务的广告商的需求。我认为这种关系可以用一个更具罪犯内涵的词来恰当地描述：监视。

监控

作为一个思想实验，尝试用监视来替换单词数据，并观察常用短语是否听起来如此好【93】。这样如何："在我们的监控驱动的组织中，我们收集实时监控流并将它们存储在我们的监控仓库中。我们的监控科学家使用高级分析和监测处理来获得新的见解。"

这个思想实验对于本书"设计监控 —— 密集型应用程序"来说是异乎寻常的争论，但我认为需要强调的话来强调这一点。在我们制作软件"吃世界"的尝试中【94】，我们已经建立了世界上迄今为止所见过的最伟大的大众监视基础设施。我们正朝着物联网迈进，我们正在迅速接近这样一个世界：每个有人居住的空间至少包含一个互联网连接的麦克风，以智能手机，智能电视，语音控制助理设备，婴儿监视器甚至儿童玩具使用基于云的语音识别。这些设备中的很多都有可怕的安全记录【95】。

即使是极权主义和专制政权也只能梦想在每个房间放置一个麦克风，并强迫每个人不断地携带能够追踪其位置和动作的设备。然而，我们显然自愿地，甚至热心地投身于这个全面监视的世界。不同之处在于数据是由公司而不是由政府机构收集的【96】。

并不是所有的数据收集都必须符合监督的要求，但检查它们可以帮助我们理解我们与数据收集者的关系。为什么我们似乎很乐意接受企业的监督？也许你觉得你没有隐瞒 —— 换句话说，你完全符合现有的权力结构，你不是被边缘化的少数派，你不必害怕迫害【97】。不是每个人都如此幸运。也许这是因为目的似乎是良性的 —— 这不是强制性的，也不是强制性的，而只是更好的建议和更个性化的营销。但是，结合上一节中对预测分析的讨论，这种区分似乎不太清晰。

我们已经看到与汽车追踪设备挂钩的汽车保险费以及取决于佩戴健身追踪设备的人的健康保险范围。当监视被用来确定在生活的重要方面如保险或就业等方面的东西时，它开始变得不那么温和。此外，数据分析可以揭示出令人惊讶的侵入性事物：例如，智能手表或健身追踪器中的运动传感器可用于以相当好的准确度计算出您正在输入的内容（例如密码）【98】。而分析算法只会变得更好。

同意和选择的自由

我们可能会断言用户自愿选择使用跟踪其活动的服务，并且他们已同意服务条款和隐私政策，因此他们同意收集数据。我们甚至可以声称，用户正在接受有价值的服务，以换取所提供的数据，并且为了提供服务，跟踪是必要的。毫无疑问，社交网络，搜索引擎以及其他各种免费的在线服务对于用户来说都是有价值的，但是这个说法存在问题。

用户几乎不知道他们提供给我们的数据库的数据，或者它们如何保留和处理 —— 而大多数隐私政策的作用更多的是模糊而不是照亮。如果不了解他们的数据会发生什么，用户无法给出任何有意义的同意。通常，来自一个用户的数据还说明关于不是该服务的用户并且没有同意任何条款的其他人的事情。我们在本书的这一部分讨论的衍生数据集（来自整个用户群的数据可能与行为跟踪和外部数据源相结合）恰恰是用户无法获得任何有意义理解的数据种类。而且，数据是通过单向过程从用户中提取出来的，而不是真正的互惠关系，而不是公平的价值交换。没有对话，用户无法选择他们提供的数据量以及他们收到的服务回报：服务和用户之间的关系是非常不对称和片面的。这些条款是由服务设置，而不是由用户【99】。

对于不同意监视的用户，唯一真正的选择就是不使用服务。但是这个选择也不是免费的：如果一项服务如此受欢迎以至于“被大多数人认为是基本社会参与的必要条件”【99】，那么指望人们选择退出这项服务是不合理的 —— 使用它事实上是强制性的。例如，在大多数西方社会群体中，携带智能手机，使用Facebook进行社交以及使用Google查找信息已成为常态。特别是当一项服务具有网络效应时，人们选择不使用它会产生社会成本。

由于跟踪用户而拒绝使用服务，这只是少数拥有足够的时间和知识来了解其隐私政策的人员的一种选择，并且有可能错过社会参与或专业人士如果他们参与了服务，可能会出现机会。对于处境不太好的人来说，没有任何意义上的自由选择：监督变得不可避免。

数据的隐私和使用

有时候，人们声称“隐私已经死了”，理由是有些用户愿意把各种有关他们生活的事情发布到社交媒体上，有时是通常的，有时是个人的。但是，这种说法是错误的，并且依赖于对隐私一词的错误理解。

拥有隐私并不意味着保密一切;它意味着要有自由选择要向谁揭示哪些事情，要公开什么，以及要保密什么。隐私权是一项决定权：它可以让每个人决定他们希望在每个场合保密和透明度之间的区域【99】。这是一个人的自由和自主的重要方面。

当通过监控基础设施从人身上提取数据时，隐私权不一定被侵蚀，而是转移到数据收集器。获取数据的公司本质上是说“相信我们用你的数据做正确的事情”，这意味着决定要透露什么和保密的权利是从个人转移到公司的。

这些公司反过来选择保留这个监视秘密的大部分结果，因为揭示它会被认为是令人毛骨悚然的，并且会损害他们的商业模式（这比其他公司更依赖于对人的了解）。关于用户的亲密信息只是间接显示的，例如以广告针对特定人群（如那些患有特定疾病的人群）的工具的形式。

即使特定用户无法从特定广告定位的人群中个人身份识别，他们已经失去了一些关于披露一些隐私信息的机构，例如他们是否患有某种疾病。决定根据个人喜好向谁透露什么的不是用户，而是公司以最大化利润为目标行使隐私权。

许多公司都有一个不被视为令人毛骨悚然的目标 —— 避免了他们的数据收集的实际侵入性问题，而是专注于管理用户感知。即使这些看法经常被糟糕的管理：例如，事实可能是事实上的正确，但如果它触发痛苦的回忆，用户可能不希望被提醒它【100】。对于任何类型的数据，我们都应该期望在某种程度上出现错误，不可取或不适当的可能性，我们需要建立处理这些失败的机制。无论是“不可取的”还是“不适当的”，当然都是由人的判断决定的;除非我们明确地规划它们尊重人类的需求，否则算法会忽略这些概念。作为这些系统的工程师，我们必须谦虚，接受和规划这些失败。

允许在线服务的用户控制其他用户可以看到的数据的哪些方面的隐私设置是将一些控制交还给用户的起点。但是，无论设置如何，服务本身仍然可以不受限制地访问数据，并且可以以隐私策略允许的任何方式自由使用它。即使服务承诺不会将数据出售给第三方，它通常会授

予自己不受限制的权利，以在内部处理和分析数据，而且往往比用户明显看到的要多得多。

这种从个人到公司的大规模隐私权转移历史上是史无前例的【99】。监控一直存在，但它过去是昂贵和手动的，不可扩展和自动化。信托关系一直存在，例如患者与其医生之间，或被告与其律师之间 —— 但在这些情况下，数据的使用严格受到道德，法律和监管限制的约束。互联网服务使得在没有有意义的同意的情况下收集大量敏感信息变得容易得多，并且在没有用户理解他们的私人数据正在发生的情况下大规模使用它。

数据作为资产和权力

由于行为数据是用户与服务交互的副产品，因此有时称为"数据耗尽" —— 表明数据是毫无价值的浪费材料。从这个角度来看，行为和预测分析可以被看作是一种从数据中提取价值的回收形式，否则这些数据会被抛弃。

更准确的说是反过来看：从经济的角度来看，如果有针对性的广告是为服务付费的话，那么关于人的行为数据就是服务的核心资产。在这种情况下，用户与之交互的应用程序仅仅是一种诱骗用户将更多的个人信息提供给监控基础设施的手段【99】。数据提取机器人讥讽地发现，在线服务中常常表现出令人愉快的人类创造力和社交关系。

个人数据是宝贵资产的说法得到了数据中介的支持，这个数据中介是一个隐蔽的行业，它主要是为了市场目的而采购，收集，分析，推断和转售侵入性个人数据【90】。初创公司通过他们的用户数量，通过"眼球"，即通过他们的监视能力来估价。

因为数据很有价值，所以很多人都想要它。当然，公司需要它 —— 这就是为什么他们收集它的原因。但政府也想获得它：通过秘密交易，胁迫，法律强制或者只是偷窃【101】。当公司破产时，收集到的个人资料就是被出售的资产之一。而且，数据难以确保，因此违规事件经常令人不安地发生【102】。

这些观察已经导致批评者说数据不仅仅是一种资产，而是一种"有毒资产"【101】，或者至少是"有害物质"【103】。即使我们认为我们有能力防止滥用数据，但是每当我们收集数据时，我们都需要平衡这些好处和落入他们手中的风险：计算机系统可能会被犯罪分子或敌对的外国情报服务，数据可能会被内部人士泄露，公司可能会落入不合法的管理层之中，而这些管理层不会分享我们的价值观，或者这个国家可能会被毫无疑问迫使我们交出数据的政权所接管。

正如古老的格言所言，"知识就是力量"。此外，"在避免审查自己的同时审视他人是最重要的权力形式之一"【105】。这就是为什么极权政府希望监督：这让他们有能力控制人口。尽管今天的科技公司并没有公开地寻求政治权力，但是他们积累的数据和知识却给他们带来了很多权力，其中大部分是在公共监督之外偷偷摸摸的【106】。

记住工业革命

数据是信息时代的决定性特征。互联网，数据存储，处理和软件驱动的自动化正在对全球经济和人类社会产生重大影响。随着我们的日常生活和社会组织在过去的十年中发生了变化，并且在未来的十年中可能会继续发生根本性的变化，所以我们就会想到与工业革命的比较【87,96】。

工业革命是通过重大的技术和农业进步来实现的，它带来了持续的经济增长，长期的生活水平显着提高。然而，它也带来了一些重大问题：空气污染（由于烟雾和化学过程）以及水（工业和人类的废物）造成的污染是可怕的。工厂老板生活在辉煌中，而城市工人经常住在非常贫困的住房里，并且在恶劣的条件下长时间工作。童工是常见的，包括在矿场工作的危险和低薪。

花了很长时间才制定了保护措施，例如环境保护条例，工作场所安全协议，禁止使用童工和食品卫生检查。毫无疑问，生产成本增加了，因为事实再也不能把废物倒入河流，销售污染的食物或者剥削工人。但是，整个社会都从中受益匪浅，我们中很少人会想要回到这些规定之前的时间【87】。

就像工业革命有一个黑暗的一面需要管理一样，我们对信息时代的转变也有我们需要面对和解决的重大问题。我相信收集和使用数据是其中的一个问题。用布鲁斯·施奈尔（Bruce Schneier）【96】的话来说：

> 数据是信息时代的污染问题，保护隐私是环境挑战。几乎所有的电脑都能生成信息。它停留在周围，溃烂。我们如何处理它 —— 我们如何控制它以及如何处理它 —— 对我们信息经济的健康至关重要。正如我们今天回顾工业时代的早期十年，并想知道我们的祖先在匆忙建立一个工业世界的过程中如何忽略污染，我们的孙辈在信息时代的前几十年将回顾我们，就我们如何应对数据收集和滥用的挑战来判断我们。
>
> 我们应该设法让他们感到骄傲。

立法和自律

数据保护法可能有助于维护个人的权利。例如，1995年的“欧洲数据保护指令”规定，个人数据必须“为特定的，明确的和合法的目的收集，而不是以与这些目的不相符的方式进一步处理”，并且数据必须“充分，与收集目的相关并不过分“【107】。

但是，这个立法在今天的互联网环境下是否有效还是有疑问的【108】。这些规则直接违背了大数据的理念，即最大限度地收集数据，将其与其他数据集结合起来进行试验和探索，以便产生新的见解。探索意味着将数据用于未预期的目的，这与用户同意的“特定和明确”目的相反（如果我们可以有意义地谈论同意的话）【109】。目前正在制定更新的规定【89】。

收集大量有关人员数据的公司反对监管，认为这是创新的负担和阻碍。在某种程度上，这种反对是有道理的。例如，分享医疗数据时，隐私存在明显风险，但也有潜在机会：如果数据分析能够帮助我们实现更好的诊断或找到更好的治疗方法，可以预防多少人死亡【110】？过度监管可能会阻止这种突破。这种潜在机会与风险之间难以平衡【105】。

从根本上说，我认为我们需要在个人数据方面进行科技行业的文化转变。我们应该停止将用户作为度量标准进行优化，并记住他们是值得尊重，尊严和代理的人。我们应该自我调节我们的数据收集和处理实践，以建立和维持依赖我们软件的人们的信任【111】。我们应该自己去教育最终用户如何使用他们的数据，而不是让他们处于黑暗中。

我们应该允许每个人保持他们的隐私 —— 即他们控制自己的数据 —— 而不是通过监视来窃取他们的控制权。我们控制数据的个人权利就像是一个国家公园的自然环境：如果我们没有明确的保护和关心，它将被破坏。这将是公地的悲剧，我们都会因此而变得更糟。无所不在的监视不是不可避免的，我们仍然能够阻止它。

我们究竟能做到这一点是一个悬而未决的问题。首先，我们不应该永久保留数据，但一旦不再需要就立即清除数据【111,112】。清除数据与不变性的想法背道而驰（请参阅第463页的“不变性的限制”），但可以解决该问题。我所看到的一种很有前途的方法是通过加密协议来实施访问控制，而不仅仅是通过策略【113,114】。总的来说，文化和态度的变化是必要的。

本章小结

在本章中，我们讨论了设计数据系统的新方法，并且包括了我对未来的个人意见和猜测。我们从观察开始，即没有一种工具可以有效地服务于所有可能的用例，因此应用程序必须编写几个不同的软件才能实现其目标。我们讨论了如何使用批处理和事件流来解决这个数据集成问题，以便让数据变化在不同系统之间流动。

在这种方法中，某些系统被指定为记录系统，而其他数据则通过转换从中得出。通过这种方式，我们可以维护索引，物化视图，机器学习模型，统计摘要等等。通过使这些衍生和转换异步和松散耦合，防止一个区域中的问题扩散到系统的不相关部分，从而增加整个系统的稳健性和容错性。

将数据流表示为从一个数据集到另一个数据集的转换也有助于演化应用程序：如果您想更改其中一个处理步骤，例如更改索引或缓存的结构，则可以在整个输入数据集上重新运行新的转换代码以便重新输出。同样，如果出现问题，您可以修复代码并重新处理数据以便恢复。

这些过程与内部数据库已经完成的过程非常相似，因此我们重新构思了数据流应用程序的概念，将数据库的组件分开，并通过组合这些松散耦合的组件来构建应用程序。

衍生状态可以通过观察底层数据的变化来更新。此外，下游消费者可以进一步观察衍生状态本身。我们甚至可以将此数据流一直传送到显示数据的最终用户设备，从而构建可动态更新以反映数据更改并继续脱机工作的用户界面。 接下来，我们讨论了如何确保所有这些处理在出现故障时保持正确。我们看到强大的完整性保证可以通过异步事件处理可视化地实现，通过使用端到端操作标识符使操作幂等并异步检查约束。客户可以等到检查通过，或者不用等待就行，但是可能会有违反约束的道歉风险。这种方法比使用分布式事务的传统方法更具可扩展性和可靠性，并且适合于实践中有多少业务流程工作。

通过构建围绕数据流的应用程序并同步检查约束条件，我们可以避免大多数协调，并创建维护完整性但仍能很好地运行的系统，即使在地理分布的情况下和出现故障时也是如此。然后，我们谈了一些关于使用审计来验证数据的完整性并检测腐败的问题。

最后，我们退后一步，审查了构建数据密集型应用程序的一些道德问题。我们看到，虽然数据可以用来做好事，但它也可能造成重大损害：作出严重影响人们生活并难以申诉的决定的正当理由，导致歧视和剥削，规范监督以及揭露私密信息。我们也冒着数据泄露的风险，并且我们可能会发现善意使用数据会产生意想不到的后果。

由于软件和数据对世界产生如此巨大的影响，我们的工程师们必须记住，我们有责任为我们想要生活的这个世界努力工作：一个对待人性与尊重的世界。我希望我们能够一起为实现这一目标而努力。

参考文献

1. Rachid Belaid: "Postgres Full-Text Search is Good Enough!," *rachbelaid.com*, July 13, 2015.
2. Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: "Challenges to Adopting Stronger Consistency at Scale," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.
3. Pat Helland and Dave Campbell: "Building on Quicksand," at *4th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2009.
4. Jessica Kerr: "Provenance and Causality in Distributed Systems," *blog.jessitron.com*, September 25, 2016.
5. Kostas Tzoumas: "Batch Is a Special Case of Streaming," *data-artisans.com*, September 15, 2015.
6. Shinji Kim and Robert Blafford: "Stream Windowing Performance Analysis: Concord and Spark Streaming," *concord.io*, July 6, 2016.
7. Jay Kreps: "The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction," *engineering.linkedin.com*, December 16, 2013.
8. Pat Helland: "Life Beyond Distributed Transactions: An Apostate's Opinion," at *3rd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2007.
9. "Great Western Railway (1835–1948)," Network Rail Virtual Archive, *networkrail.co.uk*.
10. Jacqueline Xu: "Online Migrations at Scale," *stripe.com*, February 2, 2017.
11. Molly Bartlett Dishman and Martin Fowler: "Agile Architecture," at *O'Reilly Software Architecture Conference*, March 2015.

12. Nathan Marz and James Warren: *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*. Manning, 2015. ISBN: 978-1-617-29034-3

13. Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin: "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations," at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.

14. Jay Kreps: "Questioning the Lambda Architecture," *oreilly.com*, July 2, 2014.

15. Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, et al.: "Liquid: Unifying Nearline and Offline Big Data Integration," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

16. Dennis M. Ritchie and Ken Thompson: "The UNIX Time-Sharing System," *Communications of the ACM*, volume 17, number 7, pages 365–375, July 1974. doi:10.1145/361011.361061

17. Eric A. Brewer and Joseph M. Hellerstein: "CS262a: Advanced Topics in Computer Systems," lecture notes, University of California, Berkeley, *cs.berkeley.edu*, August 2011.

18. Michael Stonebraker: "The Case for Polystores," *wp.sigmod.org*, July 13, 2015.

19. Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, et al.: "The BigDAWG Polystore System," *ACM SIGMOD Record*, volume 44, number 2, pages 11–16, June 2015. doi:10.1145/2814710.2814713

20. Patrycja Dybka: "Foreign Data Wrappers for PostgreSQL," *vertabelo.com*, March 24, 2015.

21. David B. Lomet, Alan Fekete, Gerhard Weikum, and Mike Zwilling: "Unbundling Transaction Services in the Cloud," at *4th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2009.

22. Martin Kleppmann and Jay Kreps: "Kafka, Samza and the Unix Philosophy of Distributed Data," *IEEE Data Engineering Bulletin*, volume 38, number 4, pages 4–14, December 2015.

23. John Hugg: "Winning Now and in the Future: Where VoltDB Shines," *voltdb.com*, March 23, 2016.

24. Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard: "Differential Dataflow," at *6th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2013.

25. Derek G Murray, Frank McSherry, Rebecca Isaacs, et al.: "Naiad: A Timely Dataflow System," at *24th ACM Symposium on Operating Systems Principles* (SOSP), pages 439–455, November 2013. doi:10.1145/2517349.2522738

26. Gwen Shapira: "We have a bunch of customers who are implementing 'database inside-out' concept and they all ask 'is anyone else doing it? are we crazy?'" *twitter.com*, July 28, 2016.

27. Martin Kleppmann: "Turning the Database Inside-out with Apache Samza," at *Strange Loop*, September 2014.

28. Peter Van Roy and Seif Haridi: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN: 978-0-262-22069-9

29. "Juttle Documentation," *juttle.github.io*, 2016.

30. Evan Czaplicki and Stephen Chong: "Asynchronous Functional Reactive Programming for GUIs," at *34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 2013. doi:10.1145/2491956.2462161

31. Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter: "A Survey on Reactive Programming," *ACM Computing Surveys*, volume 45, number 4, pages 1–34, August 2013. doi:10.1145/2501654.2501666

32. Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak: "Consistency Analysis in Bloom: A CALM and Collected Approach," at *5th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2011.

33. Felienne Hermans: "Spreadsheets Are Code," at *Code Mesh*, November 2015.

34. Dan Bricklin and Bob Frankston: "VisiCalc: Information from Its Creators," *danbricklin.com*.

35. D. Sculley, Gary Holt, Daniel Golovin, et al.: "Machine Learning: The High-Interest Credit Card of Technical Debt," at *NIPS Workshop on Software Engineering for Machine Learning* (SE4ML), December 2014.

36. Peter Bailis, Alan Fekete, Michael J Franklin, et al.: "Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2737784

37. Guy Steele: "Re: Need for Macros (Was Re: Icon)," email to *ll1-discuss* mailing list, *people.csail.mit.edu*, December 24, 2001.

38. David Gelernter: “Generative Communication in Linda,” *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 7, number 1, pages 80–112, January 1985. doi:10.1145/2363.2433

39. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec: “The Many Faces of Publish/Subscribe,” *ACM Computing Surveys*, volume 35, number 2, pages 114–131, June 2003. doi:10.1145/857076.857078

40. Ben Stopford: “Microservices in a Streaming World,” at *QCon London*, March 2016.

41. Christian Posta: “Why Microservices Should Be Event Driven: Autonomy vs Authority,” *blog.christianposta.com*, May 27, 2016.

42. Alex Feyerke: “Say Hello to Offline First,” *hood.ie*, November 5, 2013.

43. Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich: “Global Sequence Protocol: A Robust Abstraction for Replicated Shared State,” at *29th European Conference on Object-Oriented Programming* (ECOOP), July 2015. doi:10.4230/LIPIcs.ECOOP.2015.568

44. Mark Soper: “Clearing Up React Data Management Confusion with Flux, Redux, and Relay,” *medium.com*, December 3, 2015.

45. Eno Thereska, Damian Guy, Michael Noll, and Neha Narkhede: “Unifying Stream Processing and Interactive Queries in Apache Kafka,” *confluent.io*, October 26, 2016.

46. Frank McSherry: “Dataflow as Database,” *github.com*, July 17, 2016.

47. Peter Alvaro: “I See What You Mean,” at *Strange Loop*, September 2015.

48. Nathan Marz: “Trident: A High-Level Abstraction for Realtime Computation,” *blog.twitter.com*, August 2, 2012.

49. Edi Bice: “Low Latency Web Scale Fraud Prevention with Apache Samza, Kafka and Friends,” at *Merchant Risk Council MRC Vegas Conference*, March 2016.

50. Charity Majors: “The Accidental DBA,” *charity.wtf*, October 2, 2016.

51. Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu: “Semantic Conditions for Correctness at Different Isolation Levels,” at *16th International Conference on Data Engineering* (ICDE), February 2000. doi:10.1109/ICDE.2000.839387

52. Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan: “Automating the Detection of Snapshot Isolation Anomalies,” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

53. Kyle Kingsbury: Jepsen blog post series, *aphyr.com*, 2013–2016.

54. Michael Jouravlev: "Redirect After Post," *theserverside.com*, August 1, 2004.

55. Jerome H. Saltzer, David P. Reed, and David D. Clark: "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984. doi:10.1145/357401.357402

56. Peter Bailis, Alan Fekete, Michael J. Franklin, et al.: "Coordination-Avoiding Database Systems," *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.

57. Alex Yarmula: "Strong Consistency in Manhattan," *blog.twitter.com*, March 17, 2016.

58. Douglas B Terry, Marvin M Theimer, Karin Petersen, et al.: "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," at *15th ACM Symposium on Operating Systems Principles* (SOSP), pages 172–182, December 1995. doi:10.1145/224056.224070

59. Jim Gray: "The Transaction Concept: Virtues and Limitations," at *7th International Conference on Very Large Data Bases* (VLDB), September 1981.

60. Hector Garcia-Molina and Kenneth Salem: "Sagas," at *ACM International Conference on Management of Data* (SIGMOD), May 1987. doi:10.1145/38713.38742

61. Pat Helland: "Memories, Guesses, and Apologies," *blogs.msdn.com*, May 15, 2007.

62. Yoongu Kim, Ross Daly, Jeremie Kim, et al.: "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," at *41st Annual International Symposium on Computer Architecture* (ISCA), June 2014. doi:10.1145/2678373.2665726

63. Mark Seaborn and Thomas Dullien: "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," *googleprojectzero.blogspot.co.uk*, March 9, 2015.

64. Jim N. Gray and Catharine van Ingen: "Empirical Measurements of Disk Failure Rates and Error Rates," Microsoft Research, MSR-TR-2005-166, December 2005.

65. Annamalai Gurusami and Daniel Price: "Bug #73170: Duplicates in Unique Secondary Index Because of Fix of Bug#68021," *bugs.mysql.com*, July 2014.

66. Gary Fredericks: "Postgres Serializability Bug," *github.com*, September 2015.

67. Xiao Chen: "HDFS DataNode Scanners and Disk Checker Explained," *blog.cloudera.com*, December 20, 2016.

68. Jay Kreps: "Getting Real About Distributed System Reliability," *blog.empathybox.com*, March 19, 2012.

69. Martin Fowler: "The LMAX Architecture," *martinfowler.com*, July 12, 2011.

70. Sam Stokes: "Move Fast with Confidence," *blog.samstokes.co.uk*, July 11, 2016.

71. "Sawtooth Lake Documentation," Intel Corporation, *intelledger.github.io*, 2016.

72. Richard Gendal Brown: "Introducing R3 Corda™: A Distributed Ledger Designed for Financial Services," *gendal.me*, April 5, 2016.

73. Trent McConaghy, Rodolphe Marques, Andreas Müller, et al.: "BigchainDB: A Scalable Blockchain Database," *bigchaindb.com*, June 8, 2016.

74. Ralph C. Merkle: "A Digital Signature Based on a Conventional Encryption Function," at *CRYPTO '87*, August 1987. doi:10.1007/3-540-48184-2_32

75. Ben Laurie: "Certificate Transparency," *ACM Queue*, volume 12, number 8, pages 10-19, August 2014. doi:10.1145/2668152.2668154

76. Mark D. Ryan: "Enhanced Certificate Transparency and End-to-End Encrypted Mail," at *Network and Distributed System Security Symposium* (NDSS), February 2014. doi:10.14722/ndss.2014.23379

77. "Software Engineering Code of Ethics and Professional Practice," Association for Computing Machinery, *acm.org*, 1999.

78. François Chollet: "Software development is starting to involve important ethical choices," *twitter.com*, October 30, 2016.

79. Igor Perisic: "Making Hard Choices: The Quest for Ethics in Machine Learning," *engineering.linkedin.com*, November 2016.

80. John Naughton: "Algorithm Writers Need a Code of Conduct," *theguardian.com*, December 6, 2015.

81. Logan Kugler: "What Happens When Big Data Blunders?," *Communications of the ACM*, volume 59, number 6, pages 15–16, June 2016. doi:10.1145/2911975

82. Bill Davidow: "Welcome to Algorithmic Prison," *theatlantic.com*, February 20, 2014.

83. Don Peck: "They're Watching You at Work," *theatlantic.com*, December 2013.

84. Leigh Alexander: "Is an Algorithm Any Less Racist Than a Human?" *theguardian.com*, August 3, 2016.

85. Jesse Emspak: "How a Machine Learns Prejudice," *scientificamerican.com*, December 29, 2016.

86. Maciej Cegłowski: "The Moral Economy of Tech," *idlewords.com*, June 2016.

87. Cathy O'Neil: *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown Publishing, 2016. ISBN: 978-0-553-41881-1

88. Julia Angwin: "Make Algorithms Accountable," *nytimes.com*, August 1, 2016.

89. Bryce Goodman and Seth Flaxman: "European Union Regulations on Algorithmic Decision-Making and a 'Right to Explanation'," *arXiv:1606.08813*, August 31, 2016.

90. "A Review of the Data Broker Industry: Collection, Use, and Sale of Consumer Data for Marketing Purposes," Staff Report, *United States Senate Committee on Commerce, Science, and Transportation*, *commerce.senate.gov*, December 2013.

91. Olivia Solon: "Facebook's Failure: Did Fake News and Polarized Politics Get Trump Elected?" *theguardian.com*, November 10, 2016.

92. Donella H. Meadows and Diana Wright: *Thinking in Systems: A Primer*. Chelsea Green Publishing, 2008. ISBN: 978-1-603-58055-7

93. Daniel J. Bernstein: "Listening to a 'big data'/'data science' talk," *twitter.com*, May 12, 2015.

94. Marc Andreessen: "Why Software Is Eating the World," *The Wall Street Journal*, 20 August 2011.

95. J. M. Porup: "'Internet of Things' Security Is Hilariously Broken and Getting Worse," *arstechnica.com*, January 23, 2016.

96. Bruce Schneier: *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W. W. Norton, 2015. ISBN: 978-0-393-35217-7

97. The Grugq: "Nothing to Hide," *grugq.tumblr.com*, April 15, 2016.

98. Tony Beltramelli: "Deep-Spying: Spying Using Smartwatch and Deep Learning," Masters Thesis, IT University of Copenhagen, December 2015. Available at *arxiv.org/abs/1512.05616*

99. Shoshana Zuboff: "Big Other: Surveillance Capitalism and the Prospects of an Information Civilization," *Journal of Information Technology*, volume 30, number 1, pages 75–89, April 2015. doi:10.1057/jit.2015.5

00. Carina C. Zona: "Consequences of an Insightful Algorithm," at *GOTO Berlin*, November 2016.

01. Bruce Schneier: "Data Is a Toxic Asset, So Why Not Throw It Out?," *schneier.com*, March 1, 2016.

02. John E. Dunn: "The UK's 15 Most Infamous Data Breaches," *techworld.com*, November 18, 2016.
03. Cory Scott: "Data is not toxic - which implies no benefit - but rather hazardous material, where we must balance need vs. want," *twitter.com*, March 6, 2016.
04. Bruce Schneier: "Mission Creep: When Everything Is Terrorism," *schneier.com*, July 16, 2013.
05. Lena Ulbricht and Maximilian von Grafenstein: "Big Data: Big Power Shifts?," *Internet Policy Review*, volume 5, number 1, March 2016. doi:10.14763/2016.1.406
06. Ellen P. Goodman and Julia Powles: "Facebook and Google: Most Powerful and Secretive Empires We've Ever Known," *theguardian.com*, September 28, 2016.
07. Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data, Official Journal of the European Communities No. L 281/31, *eur-lex.europa.eu*, November 1995.
08. Brendan Van Alsenoy: "Regulating Data Protection: The Allocation of Responsibility and Risk Among Actors Involved in Personal Data Processing," Thesis, KU Leuven Centre for IT and IP Law, August 2016.
09. Michiel Rhoen: "Beyond Consent: Improving Data Protection Through Consumer Protection Law," *Internet Policy Review*, volume 5, number 1, March 2016. doi:10.14763/2016.1.404
10. Jessica Leber: "Your Data Footprint Is Affecting Your Life in Ways You Can't Even Imagine," *fastcoexist.com*, March 15, 2016.
11. Maciej Cegłowski: "Haunted by Data," *idlewords.com*, October 2015.
12. Sam Thielman: "You Are Not What You Read: Librarians Purge User Data to Protect Privacy," *theguardian.com*, January 13, 2016.
13. Conor Friedersdorf: "Edward Snowden's Other Motive for Leaking," *theatlantic.com*, May 13, 2014.
14. Phillip Rogaway: "The Moral Character of Cryptographic Work," Cryptology ePrint 2015/1162, December 2015.

术语表 【DRAFT】

> 请注意，本术语表中的定义简短而简单，旨在传达核心思想，而不是术语的完整细微之处。 有关更多详细信息，请参阅正文中的参考资料。

[TOC]

异步（asynchronous）

不等待某些事情完成（例如，通过网络将数据发送到另一个节点），并且不会假设要花多长时间。请参阅第153页上的“同步与异步复制”，第284页上的“同步与异步网络”，以及第306页上的“系统模型与现实”。

原子（atomic）

1.在并发操作的上下文中：描述一个在单个时间点看起来生效的操作，所以另一个并发进程永远不会遇到处于“半完成”状态的操作。另见隔离。

2.在事务的上下文中：即使出现故障，将一组必须全部提交或全部回滚的写入组合在一起。参见第223页的“原子性”和第354页的“原子提交和两阶段提交（2PC）”。

背压（backpressure）

强制一些数据的发送者减慢，因为收件人不能保留 与它一起。也称为流量控制。请参阅第441页上的“消息系统”。

批处理（batch process）

一种计算，它将一些固定的（通常是大的）数据集作为输入，并将其他一些数据作为输出，而不修改输入。见第十章。

边界（bounded）

有一些已知的上限或大小。例如，在网络延迟的情况下（请参阅“超时和未定义的延迟”在本页281）和数据集（请参阅第11章的介绍）。

拜占庭故障（Byzantine fault）

以任意方式表现不正确的节点，例如通过向其他节点发送矛盾或恶意消息。请参阅第304页上的“拜占庭故障”。

缓存（cache）

最近记住使用数据的组件，以加快未来对相同数据的读取速度。它通常是不完整的：因此，如果缓存中缺少某些数据，则必须从某些底层较慢的数据存储系统具有完整的数据副本。

CAP定理（CAP theorem）

一个被广泛误解的理论结果，在实践中是没有用的。参见第336页的“CAP定理”。

因果关系（causality）

事件之间的依赖关系，当一件事发生在另一件事情之前。例如，后面的事件是对早期事件的回应，或者建立在更早的事件上，或者应该根据先前的事件来理解。请参阅第186页上的“发生之前的关系和并发性”和第339页上的“排序和因果关系”。

共识（consensus）

分布式计算的一个基本问题，就是让几个节点同意某些事情（例如，哪个节点应该是数据库集群的领导者）。问题比乍看起来要困难得多。请参阅第364页上的“容错共识”。

数据仓库（data warehouse）

一个数据库，其中来自几个不同的OLTP系统的数据已经被合并和准备用于分析目的。请参阅第91页上的“数据仓库”。

声明式（declarative）

描述某些东西应该具有的属性，但不知道如何实现它的确切步骤。在查询的上下文中，查询优化器采用声明性查询并决定如何最好地执行它。请参阅第42页上的“数据的查询语言”。

非规范化（denormalize）

为了加速读取，在标准数据集中引入一些冗余或重复数据，通常采用缓存或索引的形式。非规范化的值是一种预先计算的查询结果，视图。请参见“单对象和多对象操作”（第228页）和“从同一事件日志中派生多个视图”（第461页）。

派生数据（derived data）

通过可重复的流程从其他数据创建的数据集，如有必要，您可以再次运行该数据集。通常，需要派生数据来加速对数据的特定读访问。索引，缓存和物化视图是派生数据的示例。参见第三部分的介绍。

确定性（deterministic）

描述一个函数，如果给它相同的输入，则总是产生相同的输出。这意味着它不能依赖于随机数字，时间，网络通信或其他不可预测的事情。

分布式（distributed）

在由网络连接的多个节点上运行。以部分故障为特征：系统的某些部分可能被破坏，而其他部分仍在工作，软件通常不可能知道究竟是什么被破坏。请参阅第274页上的“故障和部分故障”。

持久（durable）

以某种方式存储数据，即使发生各种故障，也不会丢失数据。请参阅第226页上的“耐用性”。

ETL（Extract-Transform-Load）

提取-转换-加载（Extract-Transform-Load）。从源数据库中提取数据，将其转换为更适合分析查询的形式，并将其加载到数据仓库或批处理系统中的过程。请参阅第91页上的“数据仓库”。

故障转移（failover）

在具有单一领导者的系统中，故障转移是将领导角色从一个节点转移到另一个节点的过程。请参阅第156页的“处理节点中断”。

容错（fault-tolerant）

如果出现问题（例如，机器崩溃或网络连接失败），可以自动恢复。请参阅第6页上的“可靠性”。

流量控制（flow control）

见背压（backpressure）。

追随者（follower）

不直接接受来自客户端的任何写入的副本，但仅处理从领导者收到的数据更改。也称为辅助，从，只读副本或热备份。请参阅第152页上的“领导和追随者”。

全文检索（full-text search）

通过任意关键字来搜索文本，通常具有附加特征，例如匹配类似的拼写词或同义词。全文索引是一种支持这种查询的次级索引。请参见第88页上的“全文搜索和模糊索引”。

图（graph）

由顶点组成的数据结构（可以引用的东西，也称为节点或实体）和边（从一个顶点到另一个顶点的连接，也称为关系或弧）。请参阅第49页上的“类似图形的数据模型”。

散列（hash）

汇集有共同点的记录。在一个记录与另一个记录有关（外键，文档参考，图中的边）的情况下最常用，查询需要获

取参考所指向的记录。请参阅第33页上的“多对一和多对多关系”和第393页上的“减少端连接和分组”。

幂等（idempotent）

索引（index）

隔离（isolation）

连接（join）

领导者（leader）

当数据或服务被复制到多个节点时，领导是被允许进行更改的指定副本。领导者可以通过某些协议选举产生，也可以由管理者手动选择。也被称为主或主。见“领导和F

线性化（linearizable）

局部性（locality）

锁（lock）

日志（log）

物化（materialize）

节点（**node**）

规范化（**normalized**）

OLAP（**Online Analytic Processing**）

OLTP（**Online Transaction Processing**）

分区（**partitioning**）

百分位点（**percentile**）

主键（**primary key**）

法定人数（**quorum**）

再平衡（**rebalance**）

复制（**replication**）

模式（**schema**）

次级索引（**secondary index**）

可序列化（**serializable**）

无共享（**shared-nothing**）

偏差（**skew**）

脑裂（**split brain**）

存储过程（**stored procdure**）

流处理（**stream process**）

同步（**synchronous**）

记录系统（**system of record**）

超时（**timeout**）

全序（**total order**）

事务（**transaction**）

两阶段提交（**2PC, two-phase commit**）

两阶段锁定（**2PL, two-phase locking**）

无限制（**unbounded**）

后记

关于作者

Martin Kleppmann是英国康桥大学分布式系统的研究员。此前他曾在互联网公司担任过软件工程师和企业家，其中包括LinkedIn和Rapportive，负责大规模数据基础架构。在这个过程中，他以艰难的方式学习了一些东西，他希望这本书能够让你避免重蹈覆辙。

Martin是一位常规会议演讲者，博主和开源贡献者。他认为，每个人都应该有深刻的技术理念，深层次的理解能帮助我们开发出更好的软件。



关于译者

冯若航

PostgreSQL DBA @ TanTan

Alibaba+-Finplus 架构师/全栈工程师 (2015 ~ 2017)

后记

《设计数据密集型应用》封面上的动物是印度野猪（**Sus scrofa cristatus**），它是在印度，缅甸，尼泊尔，斯里兰卡和泰国发现的一种野猪的亚种。它们与欧洲公猪不同，它们具有较高的背部刷毛，没有羊毛底毛，以及更大，更直的头骨。

印度的野猪有一头灰色或黑色的头发，脊椎上有僵硬的硬毛。雄性有突出的犬齿（称为T），用来与对手战斗或抵御掠食者。雄性比雌性大，但这些物种平均肩高33-35英寸，体重200-300磅。他们的天敌包括熊，老虎和各种大型猫科动物。

这些动物夜行且杂食 —— 它们吃各种各样的东西，包括根，昆虫，腐肉，坚果，浆果和小动物。野猪也被称为通过垃圾和作物田地，造成大量的破坏，并赢得农民的仇恨。他们需要每天吃4,000 ~ 4,500卡路里。公猪有着发达的嗅觉，这有助于它们寻找地下的植物材料和挖掘动物。但是它们的视力很差。

野猪在人类文化中一直具有重要意义。在印度教传说中，野猪是毗湿奴神的化身。在古希腊的丧葬纪念碑中，它是一个勇敢失败者的象征（与胜利的狮子相反）。由于它的侵略，它被描绘在斯堪的纳维亚，日耳曼和盎格鲁 ~ 撒克逊战士的盔甲和武器上。在中国十二生肖中，它象征着决心和急躁。

O'Reilly封面上的许多动物都受到威胁;所有这些对世界都很重要。要了解有关如何提供帮助的更多信息，请访问animals.oreilly.com。封面图片来自Shaw's Zoology。封面字体是URW Typewriter和Guardian Sans。文字字体是Adobe Minion Pro;图中的字体是Adobe Myriad Pro;标题字体是Adobe Myriad Condensed;代码字体是Dalton Maag的Ubuntu Mono。